NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPSNET: SOFTWARE REQUIREMENTS FOR IMPLEMENTATION OF A SAND TABLE IN THE VIRTUAL ENVIRONMENT

by

Samuel A. Kirby

September 1995

Thesis Advisor: David R. Pratt
Thesis Co-Advisor: John Falby

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Master's Thesis

4. TITLE AND SUBTITLE
NPSNET: SOFTWARE REQUIREMENTS FOR IMPLEMENTATION OF A SAND TABLE IN THE VIRTUAL ENVIRONMENT. (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Kirby, Samuel A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problem this thesis addresses is the lack of current computer applications which allow 3D graphical visualization and manipulation of abstract control measures during military planning. 3D depiction of a battle plan is needed to reduce ambiguity in planning and provide a clearer depiction of a commander's actual intent.

The approach taken was to build a networked Virtual Environment Sand Table. The system was built using NPSNET for 3D visualization and manipulation of control measures and ModSAF for the management of the measures.

The result of this thesis was the implementation of a Virtual Environment Sand Table in NPSNET. The system enables the creation and manipulation of control measures in a networked 3D Virtual Environment. The system provides intuitive visualization of control measures overlaid on Virtual Terrain. Incorporating these features into a planning tool in a Virtual Environment makes battle planning more effective and increases situational awareness by allowing expression and depiction of abstract concepts and ideas in a shared medium.

.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Virtual enviroments, distributed simulation, planning, visualization	
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

NPSNET: SOFTWARE REQUIREMENTS FOR IMPLEMENTATION OF A SAND TABLE IN THE VIRTUAL ENVIRONMENT

Samuel A. Kirby
Captain, United States Marine Corps
B.S., United States Naval Academy, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1995

Author:

_____

Samuel A. Kirby

Approved by:

_____

David R. Pratt, Thesis Advisor

_____

John Falby, Thesis Co-Advisor

_____

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problem this thesis addresses is the lack of current computer applications which allow 3D graphical visualization and manipulation of abstract control measures during military planning. 3D depiction of a battle plan is needed to reduce ambiguity in planning and provide a clearer depiction of a commander's actual intent.

The approach taken was to build a networked Virtual Environment Sand Table. The system was built using NPSNET for 3D visualization and manipulation of control measures and ModSAF for the management of the measures.

The result of this thesis was the implementation of a Virtual Environment Sand Table in NPSNET. The system enables the creation and manipulation of control measures in a networked 3D Virtual Environment. The system provides intuitive visualization of control measures overlaid on Virtual Terrain. Incorporating these features into a planning tool in a Virtual Environment makes battle planning more effective and increases situational awareness by allowing expression and depiction of abstract concepts and ideas in a shared medium.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. INTRODUCTION

As the technology of virtual environments matures, the areas in which virtual environments can be used as a paradigm to solve real world problems are also emerging. However, the use of virtual environments should not be seen as a panacea for solving all problems and the virtual environment solution should not be "shoehorned" into inappropriate problems. The user of a virtual environment should not be encumbered by the solution, rather the paradigm should enhance and ease the task at hand. A problem which meets these criteria is that of planning military operations. The military has successfully used virtual environments in simulation for training. However, there exists further potential in virtual environments to be exploited by the military in the area of *abstract visualization*.

The military has long embraced the use of simulation in training. Military aviators have utilized sophisticated, albeit expensive, simulators as an integral part of their training. Tank simulators have also been used successfully as a key component in crew training. These environments allow the simulation of situations too dangerous or too costly to be performed in the real world. These include airborne emergencies and live fire tactical exercises. These scenarios' goals are to closely simulate what would occur in the real world, to in essence make the operator feel as if he is really in the simulated situation by making the effects and stimuli of the simulation lifelike. Much of the effort in these simulations has been to create a believable fidelity of the real world. In aircraft simulators, this has included actually building physical mock-ups of cockpits and flight instruments. The other side of this effort has been to create high fidelity displays for the users of simulations to visualize the external world. The worth of such simulations is indisputable. Aircrew can respond to emergencies using correct procedures because they have been in

that situation before - in the simulator. Tank crews can successfully target and destroy enemy armor because they too have already been in that situation - in the simulator. In both cases, operators have already gained the experience needed to handle the difficult situation. Also to be noted is that while both of these situations *can* be practiced in the physical world, the simulator or virtual environment provides the means to practice the task more safely or more cheaply.

As graphics technology has progressed, a great deal of research has been put into the fidelity of the graphics scene. The efforts have included creating realistic terrain displays, realistic lighting, high fidelity image synthesis and environmental effects. These efforts have in turn produced software and hardware with performance commensurate to rendering at the needed polygon rates with robust image synthesis. Stereoscopic displays and immersive technologies such as head mounted displays and CAVE displays further increase the believability of simulation displays. A common goal of many of these research efforts is to create displays in which there is a suspension of belief by the user, that is, the fidelity of the display allows the user to concentrate on simulated tasks or experiences rather than on the display itself.

Further efforts to enhance realism include physically-based modeling to create correct object behavior. Research is also investigating modeling and efficient representation of the physical world. Efforts also include exploring channels other than the visual channel, including auditory effects, speech input and haptic interfaces. Virtual environment research has capitalized on advances in these areas and is enjoying more success in presenting increasing fidelity of the physical world in the virtual world. The entertainment industry has also benefitted from this progress with low-end virtual environments beginning to become affordable for home and personal use. However, as efforts to create increasingly realistic virtual environments proceed, efforts on finding *what* we want to represent with this bounty of fidelity must also be considered. With virtual environments the success of simulation can and should be extended, but efforts should not be *limited* to an extension and improvement of these simulations and virtual worlds. Rather, the paradigm of virtual

environments should be considered to solve a set of problems for which no other tools have existed. Virtual environments offer us a new way to think and allow us to express and display abstract and unseen thoughts.

Abstract visualization is a term trying to encompass many of the characteristics of *scientific visualization*, while also trying to broaden the areas of coverage. The National Research Council's report on Virtual Reality points out, "VE technology is a natural match for the analysis of complex, time varying datasets. *Scientific visualization* requires the informative display of abstract quantities and concepts, rather than the realistic presentation of objects in the real world [NRC95]. This statement is usually associated with data of a scientific nature or as the report also states, "Scientific visualization [MCCO87] is the use of computer graphics to create visual images that aid in the understanding of complex, often massive, numerical representations of scientific concepts or results." These methods of presenting information can be extended beyond only *scientific* concepts into other areas exhibiting data with abstract, complex characteristics. Military operations present many such problems. Problems with large complex representations, time varying datasets and abstract concepts.

Rather than being limited to presenting what would occur in the real world or an imaginary world, the virtual environment can provide an enhanced or augmented view of these worlds. While simulations provide the means to carry out operations too costly or too dangerous to perform in the real world, or to create totally imaginary worlds, virtual environments additionally provide the means for augmented capabilities which are *physically impossible* in the real world. This should not be confused with, for example, a monster in a virtual environment game which, while *physically impossible*, is still a world view of an entity or physical object. Rather, the potential which should be recognized is how the flow of thoughts and ideas can be greatly improved with the tool of virtual environments.

An example of such visualization of normally unseen entities is *Tracktur*, which enables viewing fluid flowing over a vortex. This effort produced an Immersive

Environment which allowed a viewer to track a turbulent flow [BANK95]. While in this case a turbulent flow physically exists, the virtual environment allows the visualization of the turbulence. Perhaps even more significant is the unique way in which this visualization enables the viewer to conceptualize the turbulence in a manner which no other medium could provide. Similar efforts by NASA with virtual wind tunnels exploit the virtual environment paradigm to visualize air flow over surfaces [BRYS91]. In both cases the virtual environment provides a radically new method to evaluate voluminous amounts of data.

Another area which can use the unique advantages of the virtual environment is that of military planning. The effort in solving military planning problems differs from previous simulations and virtual battlefields in that the emphasis is less on presenting high fidelity rendering of players, terrain and buildings and more on visualizing abstract concepts of the battle and the great amount of data the battlefield produces. The focus is less on game play and more on visualizing concepts which previously were seen only in the minds of the participants. This effort to bring out the intangibles of planning and viewing the battlefield is not new. Maps with terrain features have been used with overlays of illustrative symbols. These symbols depict various control measures and the units on the battlefield. Since these overlays give only a 2D depiction of the battlefield, they do little to visualize the battlefield, battle plan and the actual flow of battle. Participants viewing such an overlay each have their own interpretation of how the depicted symbols will translate into movement and flow in the real world. Participants also have to interpret mapped terrain into the 3D features which truly comprise the terrain. The result is a set of intangibles based on interpretation, with varying degrees of commonality in the perceptions.

To create a common conceptualization and visualization of a battlefield, planners have taken the 2D overlays and built 3D physical models. Military planners have long utilized Sand Tables to accomplish this. Sand Tables are detailed models, literally built from sand, which depict the terrain features of an area of interest. Onto these terrain models lines are placed to represent the various control measures of a plan. Physical models of units

involved in the plan are placed on the Sand Table. The model is then manipulated to gain situational awareness of what the plan will look like. This method still places a great deal of the visualization of the plan in the participant's mind. Especially lacking are high fidelity representations of terrain, detailed depiction of flow and movement, and temporal correctness of the model. The plans are placed on the Sand Table, but the actual execution of the plan is rather static. Further, limited viewpoint reduces terrain appreciation needed for situational awareness.

A Virtual Environment Sand Table has the unique capability of representing these dynamic and intangible factors in a common visual frame of reference. A virtual environment allows users to see control measures in relation to the terrain over which they were placed, something obviously unseen in the real world. A Virtual Environment Sand Table would allow the user to actually see the flow of large units over high fidelity virtual terrain. The user would be able to see variations of the plan from different vantage points on and above the terrain. Timing considerations could be viewed directly. Planners' understanding of the geometry of plans would be increased. Line of sight, cover and concealment issues could be directly visualized and examined. Additionally, the strides already made by virtual environments in model representation and atmospheric conditions would enhance the visualization. Perhaps most significant would be the ability to bring ideas from planners into a common forum. Decreased variance in interpretations would present less interference and information exchange would be enhanced. Lastly, an implementation of a Virtual Environment Sand Table could be placed over existing virtual environments enabling players or low level units to conduct an exercise while actually seeing the control and planning measures. Their feedback could provide planners with additional insight and players with enhanced situational awareness and understanding of the dynamics of the overall plan.

B. PROBLEM STATEMENT

The purpose of this thesis is to use a virtual environment to examine the visualization of intangible, abstract and unseen factors in a Sand Table planning problem. Emphasis is on creating a unique depiction of these factors to enhance situational awareness of planners. Control measures and traditional Sand Table techniques are presented in the virtual environment. Control measures can be placed on terrain models giving enhanced terrain appreciation of measure placement. Plans may be constructed, executed and viewed over virtual terrain.

Units are represented as icons on the Virtual Environment Sand Table. Emphasis is on depicting the flow of units on the battle field. Temporal aspects of control measures and timing aspects of moving entities are visualized, giving planners greater understanding of the dynamics of the plan. Planners are able to take virtual tours of the battlefield increasing their understanding of the geometry of the plan. They are able to view the plan as a whole from a "god's eye" view as well as tour crucial parts of the terrain to examine plan flow from varying viewpoints. Tours also resolve line of sight and visibility issues in a context where the entire plan is visualized.

C. APPROACH

In order to investigate and demonstrate the visualization of planning, a networked Virtual Environment Sand Table is constructed on top of the Naval Postgraduate School's NPSNET. NPSNET provides a rich virtual world with robust visualization of the battlefield and real-time distributed interaction in the virtual world. NPSNET provides an excellent platform on which to construct a Virtual Environment Sand Table. Much of the functionality of NPSNET is retained and utilized to effectively enhance the visualization during planning.

The Virtual Environment Sandtable is designed and built around the following guidelines:

1. System Requirements

The system is designed to run on Silicon Graphics Onyx Reality engine computers and Silicon Graphics Indigo Computers. Software is constructed using C++ and Silicon Graphics *Performer* real-time development libraries.

2. Control Measure Placement

The Virtual Environment Sand Table enables the user to place standard control measures onto the terrain which depicts the plan being constructed. These measures include boundary lines, units symbols, linear control measures, area control measures and symbols depicting battlefield flow.

3. Viewpoint

The Virtual Environment Sand Table enables the viewer to examine the plan from many different viewpoints to include a “god’s eye view” as well views from positions on the terrain. This enables users to construct large scale plans and then visually verify the finer details of the plan.

4. Planning

The Virtual Sandtable enables the user to build complex plans over the virtual terrain and have the capability of executing the plan. The planning process is interactive which allows users to examine and critique the plan as it is being build. During execution the plan is able to proceed at different rates allowing the user to examine the plan in progress. Users can also change the plan as it is being played in order to visualize planning ideas.

5. Visualization and Manipulation

In addition to using the visualization of NPSNET, the Virtual Environment Sand Table provides the user with appealing representations of the planning tools and flow of the plan. The placement of the lines is visually appealing and intuitive. Similarly, the manipulation of lines and points is intuitive and follow a common sense approach as to how an object should move when it is manipulated.

6. ModSAF Compatibility

The Sand Table is compatible with the Modular Semi-Automated Forces System which was produced by the LORAL corporation for the U.S. Army. The Sand Table is able to display and manipulate control measures created by ModSAF and ModSAF is capable of doing the same with the Sand Table measures. This requirement is both a backwards capability in that ModSAF only offers a 2D display and a utilization of ModSAF's quite extensive automated management of control measures. ModSAF will be discussed in more detail later in Chapters II and III.

7. Networked Capability with Distributed Sand Tables

The Sand Table is able to operate on a single workstation, or work in concert with another workstation running a Sand Table. Objects on one Sand Table are able to be viewed and manipulated by another Sand Table. This requirement offers a new and exciting means for geographically separated units to engage in interactive planning. No medium has offered this capability which results in the reduction of ambiguity and increase in coordination.

D. SUMMARY

The virtual environment enables developers to go beyond creating high fidelity representations of physical and visual objects of the real and imaginary worlds. Virtual environments enable the user to actually visualize abstract concepts, dynamic ideas and physical phenomena which would normally be unseen. This thesis uses this capability as a paradigm to investigate the visualization of creating complex battlefield plans on a Virtual Environment Sand Table.

II. BACKGROUND

A. INTRODUCTION

While *abstract visualization* is an ongoing research topic, the area of planning visualization is a new area of research as is each new application using *abstract visualization*. The Sand Table is actually comprised of individual components on which research has been conducted. These components include those efforts in *abstract visualization* which have explored scientific, medical, computation and data visualization. Another area of research has been in providing interactive planning tools. Lastly, research has been conducted in the development of large scale synthetic battlefields. Each of these components will be briefly examined below with examples of research in each area.

B. ABSTRACT VISUALIZATION

1. Scientific Visualization

a. Turbulence Tracking

As mentioned in Chapter I, the *Tracktur* system is an immersive interactive 3D system which allows researchers to visualize and analyze fluid dynamics. This effort produced an Immersive Environment which allows a viewer to track a turbulent flow. While in this case a turbulent flow physically exists, the virtual environment allows the visualization of the turbulence. Perhaps even more significant is the unique way in which this visualization enables the viewer to conceptualize the turbulence in a manner which no other medium could provide. The virtual environment provides a radically new method to evaluate voluminous amounts of data. The system visualizes gigabytes of scientific data generated from supercomputer models. The visualization allows fluid dynamicists to navigate the simulation visually, providing new insights into the flow model. [BANK95]

b. Virtual Wind Tunnel

NASA Ames has developed a *Virtual Wind Tunnel* to visualize Computational Fluid Dynamics Codes which have very large, complicated and dynamic datasets. The system uses a boom mounted display from Fake Space Systems. The system enables scientists to interact with data generated from supercomputers allowing the user to actually see and interact with air flow over aircraft surfaces. The system enabled research which previously would have had to be conducted by examination in a physical wind tunnel. The enabling power of visualizing the abstract data is apparent. [BRYS91]

c. Molecular Synthesis

Research at the University of North Carolina has allowed chemists to manipulate drug molecules in a virtual environment. Their research includes the use of haptic displays in which researchers can experience the force feedback of molecular forces in molecular docking [KALA93]. This exemplifies the display of factors which prior to being visualized in a virtual environment could only be conceptualized in a chemist's mind. This underscores a characteristic of abstract visualization in that many of the abstract ideas visualized have in fact been visualized *before*--in peoples' minds. This did not allow scientists to express ideas without interference from personal perceptions and varying experience levels. This concept is a strong undercurrent in utilizing abstract visualization for military purposes.

d. Sabot Discard Studies

A final example of scientific visualization will illustrate a potential benefit of abstract visualization. Research conducted by ARL examined Kinetic Energy penetrators (sabot rounds fired from tank cannons) to see if a sabot carrier would interfere with the sabot round after firing. While a virtual environment was not used, a visualization of data collected when firing the rounds provided interesting results. A movie was created of the visualization showing an interaction of pressures as the round was being fired. The movie

illustrated a phenomena that was not known until the visualization was produced. This example shows how visualization can provide unexpected insight into a problem.

2. Medical Visualization

Researchers at the German Research Center for Computer Technology have developed a system to visualize ultrasound examinations of the heart. The system is also used to plan surgery. Medical visualization is similar to battlefield training in that both physicians and military planners are trained to mentally visualize 3D information from 2D sources. For example, physicians mentally visualize x-rays and CT images while military planners visualize maps and overlays. Both rely heavily on training, skill and experience. The system provides a visualization of the beating heart and blood flow within the heart enabling better understanding of the dynamics of the heart. An innovative "workbench" display in which stereoscopic images are projected onto the surface of a table provides a very realistic visualization of structure and dynamics previously unseen. [KRUG94]

Other research efforts include visualization of patients' CT scans for use by neurosurgeons prior to operations. Visualization systems integrate a series of CT slices, something normally done mentally by the neurosurgeon. This allows greater localization of intracranial targets prior to the operation. The system also offers features of image rotation, zoom and translucency for examinations of structures [KALA93]. This ability to obtain "impossible viewpoints" and orientations is another feature of abstract visualization which should be stressed when used in military applications.

A final example of medical visualization is work conducted by Henry Fuchs at UNC. This effort combines a see-through head-mounted display with ultra-sound visualization to allow a physician to "see-through" a patient in real time. This research would give surgeons much greater awareness during surgical procedures. This combination of abstract visualization and augmented reality could have military applications. An example would be a see-through display for soldiers which would provide navigation displays super-imposed on the real world.

3. Data Visualization

Research by Xerox PARC using *Cone Trees* has enabled the visualization of hierarchial information structures. The efforts address the limatation's of human cognitive abilities when dealing with large scale and complex information spaces. Data spaces are represented as 3D cones which greatly increase the users ability to conceptualize and understand the data. This system has been demonstrated using a directory in a Unix file system in which 600 directories and 10,000 files were visualized at one time. Also demonstrated was the visualization of company organizational charts. The system provides an impressive visualization of very complex data spaces allowing greater understanding of the represented space by the users of the system. [ROBE91]

4. Abstract Visualization in Clinical Psychology

Researchers in Japan utilized a virtual environment to implement Sandspiel, a technique used in the diagnosis and treatment of autistic patients. Researchers created a Virtual Sand Box in which patients created landscapes which were then evaluated by doctors. Users of the system manipulated a wand to modify the virtual landscape. Placement of objects and modifications of the landscape provide visualization of abstract concepts and impressions. This effort also addresses some of the input and output concerns of creating such a system. This system addresses methods on how to express ideas and abstract notions in a shared environment. [KIJI94]

5. Computation Visualization

Researchers at the Georgia Institute of Technology are using 3D visualization to examine algorithms, computer programs and computations [STAS92]. The research addresses new methods to examine a computer program's or process' execution using 3D visualization. While this effort examines the animation of software processes rather than implementing a virtual environment, it illustrates how visualization can be used as a new and innovative paradigm to solve old problems. Further, it also shows the recurring theme that program visualization was occurring before--in the programmer's mind. The

visualization provides a means to manifest complicated concepts from a common viewpoint.

C. PLANNING TOOLS

1. Interactive Design, Analysis, and Illustration of Assemblies

Research at the University of Utah has used interactive graphical tools in the design of mechanical parts and assembly planning. The project enabled rapid visualization of parts during the design process. Top down and bottom up design of complex assemblies was made easier with the tool. The tool allowed designers to explore the parts and assembly as the parts were being designed. Conflicts between mating parts could also be discovered using a natural interface. [DRIS95]

2. ModSAF

The LORAL corporation has developed the Modular Semi-Automated Forces System for the U.S. Army. In addition to providing Semi-Automated Forces, ModSAF provides GUI based mission planning. Additionally, ModSAF forces have been implemented in the Naval Postgraduate School's NPSNET [MOHN94]. While ModSAF provides for interactive mission planning and management of units, it does not provide the visualization of the control measures and dynamic flow of a plan. Further, the product does not allow the virtual touring of the 3D battlefield. The system does provide semi-autonomous forces which would be very beneficial during planning and examining the dynamics of a battle plan in the Virtual Sandtable.

D. LARGE SCALE SYNTHETIC BATTLEFIELDS

1. Air Force Institute of Technology Virtual Environments

a. Satellite Modeler

The *Satellite Modeler* is a virtual environment application which models the near-Earth space environment and displays satellite orbits. It visualizes the spatial relations

between vehicles in orbit [STYT95]. This is an example of a situation in which the desired viewpoint is impossible to obtain physically, yet by employing a virtual environment a visualization is produced which greatly increases the ability to cognize the complex information.

b. Synthetic Battle Bridge

Another virtual environment which AFIT has created is an immersive command observatory for viewing a large area battlefield. The *Synthetic Battle Bridge* is a system designed to give the battlefield commander greater situational awareness of a battle in progress [ROHR94]. The project visualizes atmospheric effects as well as radar displays and space displays. This project does much to visualize the flow of the battle. The system provides tools which enable the user to view aggregate information about the battlefield environment. Some of the features include multiple viewpoints and visual cues to identify actors in the virtual environment. These cues include transparent "bubbles", aircraft trails and locators which give information concerning type and motion of players on the battlefield.

The goals of the *Synthetic Battle Bridge* and *Satellite Modeler* typify the difficulties and also the potential of abstract visualization applications for the military. Both offer high degrees of complexity in large environments with a large number of actors and dynamic unpredictable actions [STYT95]. Stytz further points out a key aspect of the projects which is the fact that in both projects it is difficult to maintain a mental model of the environment. This would be true in many military applications of abstract visualization. Both the *Synthetic Battle Bridge* and the *Satellite Modeler* are good starting points in the development of abstract visualization systems for the military.

2. Naval Postgraduate School NPSNET

NPSNET is a networked, distributed simulation system which allows multiple users to share the same virtual environment [BARH94]. The system evolved from a military vehicle simulator into the current virtual environment. While the system does not employ

abstract visualization it does offer great potential as a platform on which to put an abstract visualization application. This is the focus of the Virtual Environment Sand Table. This project will be discussed as an example of the direction abstract visualization applications could go.

E. SUMMARY

Each of the above visualization systems quantifies and depicts unseen, vague and complex data visually and displays it in a manner which acts as an enabling tool for greater understanding of the data. The systems offer an increased capability to the user, be it the ability to see air flow, see through a patient or see satellites in orbit. Each of the systems also replaces a process or paradigm which placed the burden of conceptualizing the data on the ability of people to visualize the data mentally. The emphasis here is not that people should not think, rather that a common visualization should stimulate a common frame of reference between people for the exchange of ideas. The visualization should be a starting point in the person's minds for *more* thought.

The major contrasts between these examples and potential military uses are the exact nature of the data being represented. Whereas the primary focus of previous efforts has been in visualizing large numeric datasets, often generated by super computers, the Sanspiel, Cone Trees and design of assemblies illustrate this is not always the case. Often the visualization simply amplifies an aspect of human perception of an idea. Further, the *Synthetic Battle Bridge* shows how the battlefield can be visualized.

III. SAND TABLE COMPONENT OVERVIEW

A. INTRODUCTION

In order to be effective, the Sand Table system must allow the user to create entities such as lines and points. These points and lines are the abstract concepts to be represented in the Virtual Environment and symbolize the control measures of the battlefield. They are also icons already in the lexicon of military planners' minds. Since these symbols are already familiar to planners, the changes made to them were only those necessary to place them in a 3D world. The problem of representing the measures in the Virtual Environment consists of two parts. The first part is developing the means to actually visualize, create, select and move the control measures.

The second aspect concerns how the objects are maintained once they are created and how they are disseminated to distributed Sand Tables. The maintenance of the measures include network protocols to enable other planning workstations to be aware of planning measures. Protocols also enable transmitting of control measures being placed and the current placement and characteristics of all the planning measures present. This maintenance is accomplished, for example, by sending a protocol data unit (PDU) on the network to indicate a measure has moved, changed color or changed type of measure. This maintenance also includes a need for periodic sending of a heartbeat message which includes all measures present.

The philosophy used in the design of the Sand Table system was to incorporate as much functionality from existing systems into the Sand Table. The two parts described above were both formidable endeavors in themselves. However, by incorporating current capabilities from existing systems, the requirements for the Sand Table were obtained. The major systems used were ModSAF and NPSNET. ModSAF can be used as a battlefield planner. It allows the user to construct plans and actually have the plans executed.

ModSAF also manages abstract measures, such as points and lines, drawn on the battlefield during a simulation. The Sand Table uses ModSAF to actually manage entities. In essence the Sand Table has some very basic capabilities to maintain control measures but whenever possible the Sand Table passes the management of the control measures to ModSAF.

Similarly, an effort to create the graphical visualization of the Sand Table from scratch would have been prohibitive. So, for the visualization and manipulation of the control measures NPSNET is used. NPSNET provides a rich ability to represent many terrain databases. NPSNET also provides the ability to navigate over terrain and "tour" an entire virtual battlefield. NPSNET has the further advantage of being written using Silicon Graphics' *Performer* which provides a very extensible application.

B. MODSAF OVERVIEW

ModSAF is comprised of both a front end and back end system. The back end of the system is SAFSim which is responsible for simulating units and sending ModSAF simulation messages to the network. SAFSim is an extensive system to both manage created entities and simulate semi-automated forces. The front end is a 2D graphical user interface (GUI) at a ModSAF station which allows the user to control the semi-automated forces and plan in the ModSAF environment. [SAFF93]

1. 2D Overlay Display

ModSAF has the capability of creating a 2D GUI to represent an *overlay* of a battleplan. The term overlay is descriptive in that it originally referred to a piece of transparent plastic laid over a map of terrain onto which a battle plan was physically drawn. Control measures were depicted symbolically on the plastic with control points and lines being drawn over the terrain on the map which represented the actual terrain. ModSAF uses this principle and presents overlays in an OSF/Motif based interface. The ModSAF GUI can be compared to the physical overlay as the Sand Table in the Virtual Environment can be compared to a physical Sand Table.

The user can build battleplans by selecting control measures and placing them on the 2D terrain of the ModSAF overlay. The control measures are selected by choosing the button for the desired measure from the vertical menu bar of the ModSAF display. This menu bar is shown in Figure 1.

The GUI has the capability of maintaining and displaying many overlays. ModSAF displays only those overlays selected by the user; however, the user can select more than one overlay. Measures sent in PDUs contain information as to which overlay they belong. This will be discussed in the Persistent Object Protocol; however, it should be noted that the existence of overlays and how ModSAF displays them is important when integrating the Sand Table system to be compatible with ModSAF.

Within the ModSAF GUI is the capability to create control measures to include points, lines, minefields and units. The user selects what type of measure he wishes to place on terrain and then uses the mouse to select the geographical position onto which the object will be placed. This is shown in Figure 2.

Once placed the user can move the measures, be they lines, points or minefields, to any location on the map display. The user can also select measures to change their color, style, thickness or other attributes. A menu for changing the attributes of a point is shown in Figure 3. Measures placed on an overlay at a ModSAF station and any subsequent changes to those measures will be displayed on all other ModSAF stations displaying the chosen overlay.

In addition to being able to place control measures, the ModSAF GUI allows for the placement of military units onto the map. Units are also selected from the vertical menu bar and positioned using the mouse. The user is given a choice as to what type the unit will be (e.g. M1 tank unit, AH-64 helicopter unit) and the size of the unit (e.g. company, battalion, section). The units can then be tasked extensively for moving on routes, attacking enemy targets, proceeding to refueling points or orbiting, to name a very small set of examples of tasking available. The types of units available and the realistic flexibility of tasking and executing missions is very inclusive for military missions and events likely

Figure 1: ModSAF GUI

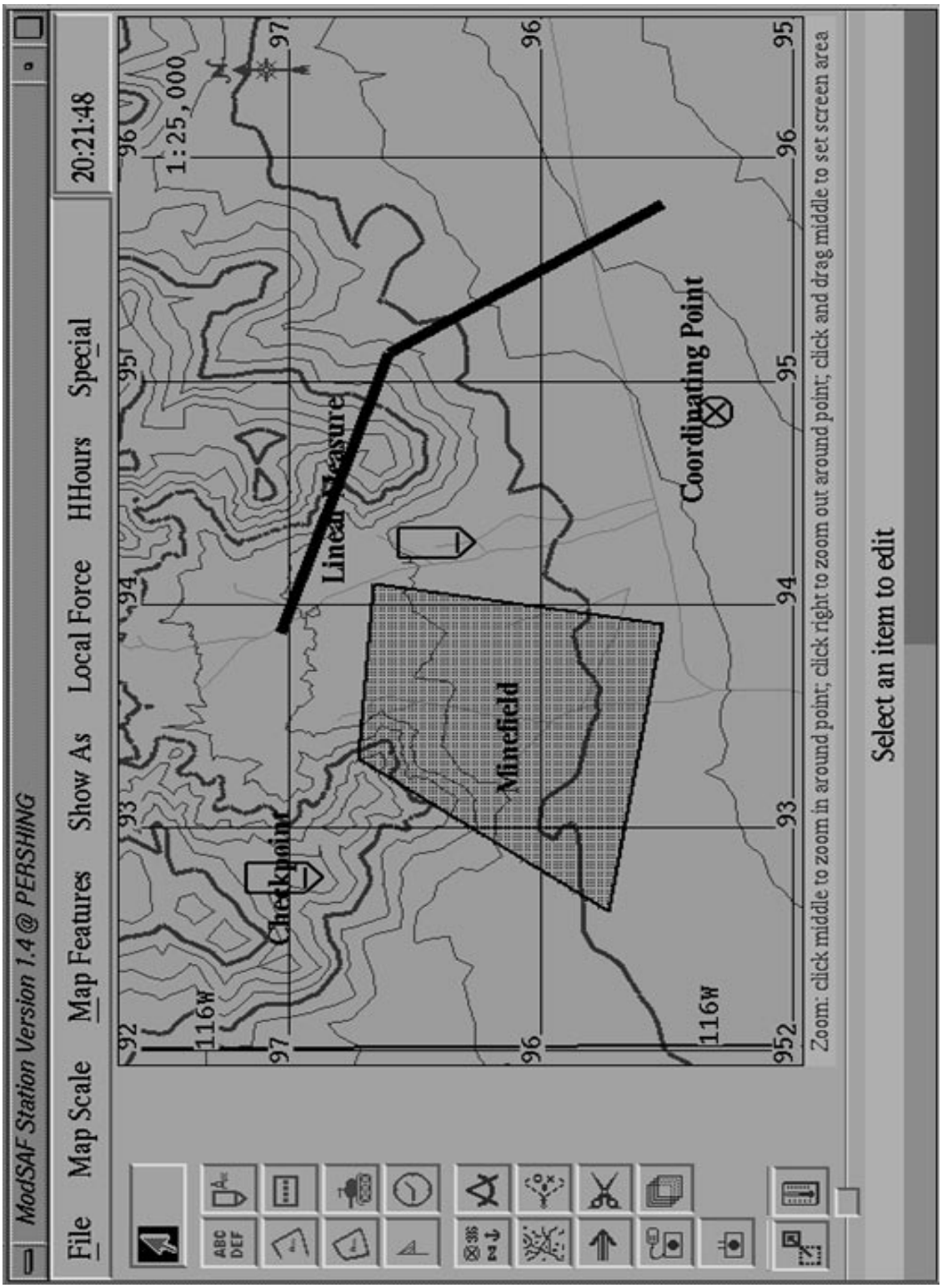


Figure 2: Placement of Control Measure in ModSAF

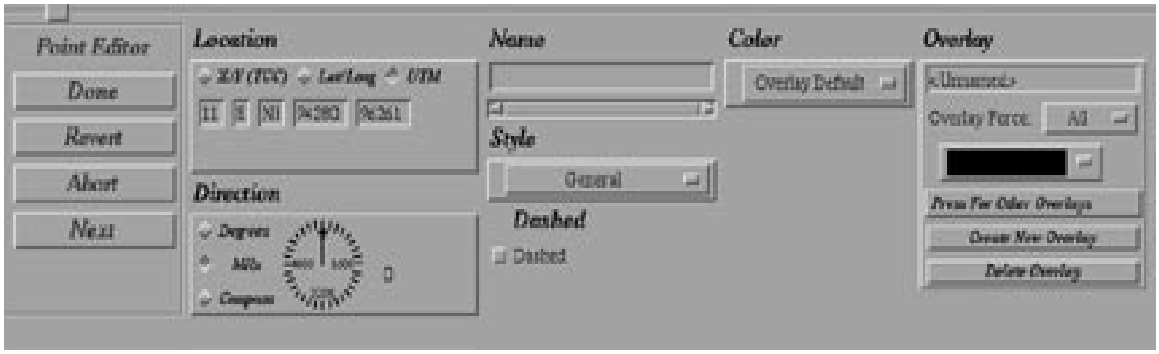


Figure 3: Control Measure Modification Menu

on a current battlefield. Further, these units can be assigned from opposing forces and will execute realistic reactions against enemy forces based on real world capabilities and assigned missions and rules of engagement. The units behave realistically in a semi-automated mode and are controlled by the mentioned back end of the ModSAF simulator.

The capabilities of ModSAF were used in NPSNET prior to the Sand Table but for a different purpose. In previous versions of NPSNET, ModSAF was used to populate the battlefield with vehicle entities. The entities moved realistically over the terrain and behaved as the vehicles they represented. The inclusion of ModSAF entities in the NPSNET battlefield proved very successful by increasing the number of vehicle entities which could be fired upon, and fire back, when the number of human players in the Virtual Environment was relatively low. However, these efforts diverge from the intent of the Sand Table in that the Sand Table's focus is less on game play and more on planning.

The Sand Table is an extension of the ModSAF GUI into the 3D Virtual Environment of NPSNET. The extension is a logical one in that the actions of creating and modifying control measures discussed above on the 2D GUI, are available on the Sand Table. However, the Sand Table is not only an extension of the GUI, rather, it is integrated with ModSAF in that entities created on one system can be displayed and manipulated on the other. Lastly, the realistic control and management of measures which are accomplished

by the ModSAF back end can be exploited by the Sand Table. Key in this integration is communication between ModSAF and the Sand Table. Fortunately, ModSAF already has complete means to communicate between simulators. This means is the Persistent Object Protocol and it is what the Sand Table uses to communicate with other Sand Tables as well as communicate with ModSAF.

2. Persistent Object Protocol

The Persistent Object (PO) Protocol is a user level network communication protocol used for communication between ModSAF simulators. The ModSAF library is written in the C programming language. The specification for the protocol is contained in LibPO, Persistent Object Library, ModSAF B Software Documentation and will be summarized [SMIT93]. ModSAF refers to workstations using the PO Protocol as *simulators* and that term will be used in this context similarly, i.e. a workstation which is using the protocol which may either be a ModSAF station or a Sand Table. What has been referred to thus far as control measures to include points, lines and minefields in the Sand Table, are a subset of what ModSAF terms Persistent Objects. Overlays are also Persistent Objects. Additionally, the PO protocol contains objects such as units, tasks, text and fire parameters. Throughout this thesis the terms “control measures” and “persistent objects” can be considered synonymous. Subtle differences do exist but they are primarily that “control measures” refers to a more abstract concept of what the Sand Table is trying to express and visualize and “persistent objects” are an implementation of those concepts in a formal specification in ModSAF.

The following is a discussion of the PO Protocol and PDUs used by the Sand Table. The protocol is described in detail with specific Sand Table use of the protocol being deferred except in cases where Sand Table usage enhances clarity. All code for described PDUs and variants can be found in the ModSAF library file p_po.h.

a. Persistent Object PDU

Communications concerning Persistent Objects are sent using Persistent Object PDUs which will be described in detail. The code excerpts for the structure definition of the Persistent Object PDU is given in Figure 4.

```
typedef struct {
      SP_PersistentObjectProtocolVersion version;
      SP_PersistentObjectPDUKind kind;
      SP_ExerciseID exercise;
      SP_DatabaseID database;
      unsigned short length;
      unsigned short _unused_40;
      union {
            SP_SimulatorPresentVariant simulatorPresent;
            SP_DescribeObjectVariant   describeObject;
            SP_ObjectsPresentVariant   objectsPresent;
            SP_ObjectRequestVariant    objectRequest;
            SP_DeleteObjectsVariant    deleteObjects;
            SP_SetWorldStateVariant    setWorldState;
            SP_NominationVariant       nomination;
      }  variant;
}  SP_PersistentObjectPDU;
```

Figure 4: Persistent Object PDU

Included in the header portion of the PDU is `version`, which contains the ModSAF protocol version being used by the sending simulator. Protocol versions are identified by date and are also contained in p_po.h. `kind` is used to identify which type of information is in the `variant` portion of the PDU. `exercise` and `database` uniquely identify the problem space which the PDUs are describing. As was mentioned, ModSAF can also be used to extensively simulate semi-automated entities on the battlefield. These simulations also utilize an exercise identifier. When a simulator is used for both simulation of entities and POs, it is expected that the exercise identifier be the same. The database identifier can be used to further subdivide an exercise into separate independent PO databases; however, currently the Sand Table utilizes only one database and the identifiers

are used by the Sand Table to ensure that both it and ModSAF are referring to the same planning space.

The `variant` portion of the PDU is the part most utilized by the Sand Table. It is the portion which describes and allows manipulation of objects between Sand Table and ModSAF workstations. In particular, three of the variants, `simulatorPresent`, `deleteObjects` and `describeObject` are used.

b. Simulator Present PDU

By definition of the PO Protocol, each simulator on the network which is interacting with other simulators using the PO Protocol broadcasts a Simulator Present PDU every 20 seconds. If this heartbeat message is not received for 48 seconds the simulator with the lightest load will take ownership of the objects owned by the simulator which failed to pass the heartbeat. The structure of the Simulator Present variant is given in Figure 5.

```
typedef struct {
      SP_SimulationAddress  simulator;
      SP_SimulatorType      simulatorType;
      unsigned short        _unused_3;
      unsigned long         databaseSequenceNumber;
      unsigned long         load;
      float                 simulationLoad;
      unsigned long         time;
      unsigned long         packetsSent;
      unsigned short        unitDatabaseVersion;
      unsigned              relativeBattleScheme   : 1;
      unsigned              _unused_4              : 15;
      SP_TerrainDatabaseID  terrain;
      char                  hostname[SP_maxSPHostnameLength];
} SP_SimulatorPresentVariant;
```

Figure 5: Simulator Present Variant PDU

The Simulator Present PDU contains many fields for an extensive protocol which includes protocols for nomination of simulators to assume the loads of simulators

not responding, load balancing and synchronization. Maintaining the stated design philosophy, the Sand Table tries to shed all of the PO maintenance to ModSAF stations and as such does not assist in the load balancing or assumption of ownership of objects created by other simulators. The Sand Table does not utilize all of the ModSAF functionality and only those fields of `SP_SimulatorPresentVariant` which relate to functionality used by the Sand Table will be discussed. `simulator` is comprised of the site and host identification of the simulator. `simulatorType` describes what type simulator has sent the simulator present PDU. More specifically, the type of simulator allows ModSAF stations to determine the capabilities of other simulators for possible load balancing. The types of simulators are contained in basic.h which is contained in the ModSAF library. The Sand Table uses a type of `SP_simulatorUnknown` since no other simulators would recognize a Sand Table type simulator at this time.

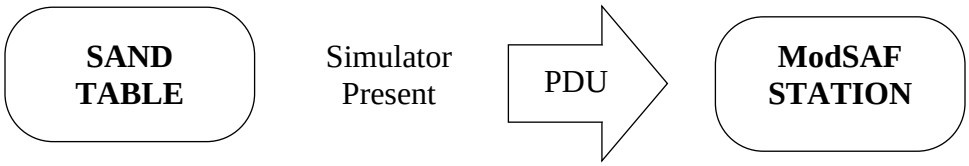
`databaseSequenceNumber` while not specifically used by the Sand Table, is an important field to coordinate with ModSAF in order to ensure the proper maintenance of measures by ModSAF. `databaseSequenceNumber` is the identity of the shared database. It is used to delete all of the objects from a database. As defined by the PO Protocol each simulator maintains the current sequence number of its database. If a simulator receives a sequence number higher than its current sequence number, it must remove all objects from its database and increase its database sequence. If a simulator receives a sequence number lower than its own, it should send a Simulator Present PDU to ensure that the simulator with the lower sequence number receives the *proper* sequence number. A simulator should start with a sequence number of zero and increment accordingly on receipt of higher sequence numbers.

As was stated, the Sand Table does not explicitly *use* the capability to delete all of the objects in the database. However, the Sand Table must maintain what the database sequence number is, or ModSAF may delete objects with a lower sequence number. If the objects are created on the Sand Table and broadcast with an invalid database sequence number, the objects will *not* be maintained by the ModSAF simulators.

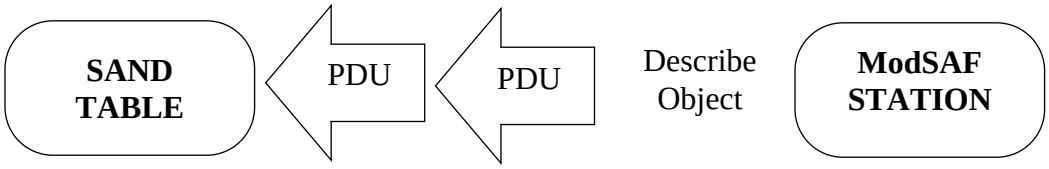
`terrain` is used to specify the identity of the terrain database being used. This is needed by ModSAF for the representation of terrain on the 2D GUI display. NPSNET uses a different terrain database format; however, the database is needed by the Sand Table for registration and proper placement of lines which are created on a ModSAF station. In particular, a database file is needed when a line segment which follow roads is sent from a ModSAF station. A ModSAF library program is used to make the conversion from road segments into points and the terrain database is used.

The `SP_SimulatorPresentVariant` is used by the Sand Table primarily for two purposes. As specified in the PO Protocol definition, when a Simulator Present PDU describes a new object, receivers should start a periodic transmission of Describe Object PDUs. The is one of the primary uses of Simulator Present PDUs by the Sand Table. When a Sand Table is initially run, it will transmit a Simulator Present PDU. Simulators maintaining persistent objects will then begin broadcasting Describe Object PDUs enabling the Sand Table to learn the objects present in the current planning problem.

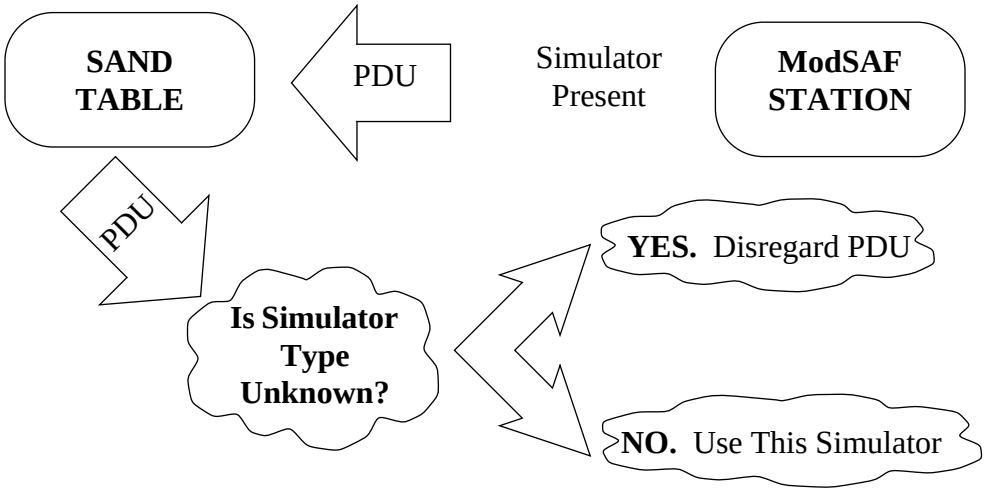
The second use of the `SP_SimulatorPresentVariant` is used in the passing of persistent object maintenance to ModSAF stations. When a Sand Table receives a `SP_SimulatorPresentVariant` PDU, the PDU is checked by examining the type to ensure the simulator present is not another Sand Table. If the `simulatorType` is of type `SP_simulatorUnknown`, the Sand Table assumes that the simulator present was from another Sand Table and the simulator is not captured. If it is not, the receiving Sand Table will then use the sending simulator to manage objects which are created or modified by the Sand Table. This will be discussed in greater detail; however, the importance of the `SP_SimulatorPresentVariant` PDU should be noted. A summary of the use of the `SP_SimulatorPresentVariant` PDU is given in Figure 6. Lastly, the Sand Table does transmit a heartbeat; however, since ownership and maintenance of persistent objects is shed to other simulators, the absence of this heartbeat from Sand Tables at this time would not have an effect.



Upon Start-up



ModSAF Response



Sand Table Response

Figure 6: Use of Simulator Present PDU

c. Delete Object PDU

Another variant of the `SP_PersistentObjectPDU` is the `deleteObjects` variant. The PDU is very straightforward and is sent by a simulator to remove persistent

objects from the appropriate database. As specified in the protocol, a deleting simulator must be able to rebroadcast a Delete Object PDU in response to Describe Object PDUs pertaining to an object which was deleted. This rebroadcast of the Delete Object PDU continues for five minutes thereby ensuring that all simulators maintain an accurate representation of the current PO database. The structure is given in Figure 7.

```
typedef struct {
      SP_SimulationAddress            deletingSimulator;
      unsigned char                   objectCount;
      unsigned                        _unused_38      :24;
      SP_ObjectIDWorldStateIDPair     objects[1];
}  SP_DeleteObjectsVariant;
```

Figure 7: Delete Objects Variant PDU

`deletingSimulator` includes the site and host of the simulator which is deleting the objects. It is not used explicitly by the Sand Table; however, when Delete Object PDUs are sent, the site and host of the Sand Table are appropriately included for use by ModSAF simulators. Of primary importance in the Delete Objects variant is `objectCount` and `objects[1]`. These fields in the structure are the array of the identities of the objects to be deleted and the actual number of those objects. The objects in the array are identified by a `SP_ObjectIDWorldStateIDPair`. Contained within the `SP_ObjectIDWorldStateIDPair` is an `objectID` which uniquely identifies a Persistent Object. This identification is comprised of the site and host of the simulator at which the object was created and an object number which is unique and incremented for each new object at the creating simulator. Removal of the objects specified in the Delete Object PDU are then handled locally by each simulator with the deleting simulator rebroadcasting the Delete PDU as necessary in response to errant simulators sending Describe Object PDUs for deleted objects.

d. Describe Object PDU

The final variant of the `SP_PersistentObjectPDU` is the `describeObject` variant and it is used to create new objects, modify old objects and change the owner of objects. `describeObject` is the most important PDU variant used by the Sand Table and it carries most of the functionality of the Sand Table. As the name implies the PDU is used for the description of persistent objects in the desired exercise and database. More specifically, the PDU variant contains the class of the objects described and all of the characteristics of the object. The Describe Object PDU contains a sub-protocol used for the different classes of persistent objects. The sub-protocol is contained in the variant portion of the `SP_DescribeObjectVariant` structure. The code for the Describe Object variant is given in Figure 8.

As with the other PDU variants discussed, the Sand Table does not use much of the functionality supported by the Describe Object PDU and, as such, only the pertinent parts will be examined in detail. `databaseSequenceNumber` is the identity of the shared database and was discussed with the Simulator Present PDU. Its importance should be reiterated in the Describe Object PDU. In order for a ModSAF simulator to maintain a persistent object created by a Sand Table, the `databaseSequenceNumber` must not be less than the current database sequence number maintained by the ModSAF station.

`objectID` uniquely identifies each object in the persistent object database and is the same identifier contained in the Delete Object PDU. It consists of the creating simulator site and host as well as a unique object number for that simulator. `worldStateID` is not used by the Sand Table; however, it has potential for future versions and added capabilities with the Sand Table. The world state enables the representation of objects in different times or states of the problem. This would enable the display of future or past objects on the Sand Table or ModSAF overlay. Objects can be represented in other world states by changing the `worldStateID` in the PDU. Currently, the Sand Table uses only the Real Time World State which is represented by zero in the site, host and object of the `worldStateID`.

```
typedef struct {
   unsigned long             databaseSequenceNumber;
   SP_ObjectID               objectID;
   SP_ObjectID               worldStateID;
   SP_SimulationAddress      owner;
   unsigned short            sequenceNumber;
   SP_PersistentObjectClass  mclass;
   unsigned                  missingFromWorldState :1;
   unsigned                  _unused_35            :7;

   union {
       SP_WorldStateClass              worldState;
       SP_OverlayClass                 overlay;
       SP_PointClass                   point;
       SP_LineClass                    line;
       SP_SectorClass                  sector;
       SP_TextClass                    text;
       SP_UnitClass                    unit;
       SP_StealthControllerClass       stealthController;
       SP_HHourClass                   hHour;
       SP_TaskClass                    task;
       SP_TaskStateClass               taskState;
       SP_TaskFrameClass               taskFrame;
       SP_TaskAuthorizationClass       taskAuthorization;
       SP_ParametricInputClass         parametricInput;
       SP_ParametricInputHolderClass   parametricInputHolder
       SP_ExerciseInitializerClass     exerciseInitializer;
       SP_FireParametersClass          fireParameters;
       SP_MinefieldClass               minefield;
       SP_SimulationRequestClass       simulationRequest;
}  variant;
}  SP_DescribeObjectVariant;
```

Figure 8: Describe Object Variant PDU

`owner` and `sequenceNumber` are the most important fields in the Describe Object PDU for the shedding of PO management to ModSAF stations. The owner consists of the site and host of the simulator which currently owns the object. Note that the owner does *not* have to be the simulator which created the object. The `sequenceNumber`, not to be confused with `databaseSequenceNumber`, is a number which is incremented each time an object is changed. Initially, when an object is created it has a sequence number of one. When the owner of an object wishes to change an object, it increments the sequence

number and sends a new Describe Object PDU with the increased sequence number as well as the modified fields which describe the object.

To create a Persistent Object, a simulator sends a describe object PDU. By the PO Protocol, the owner then transmits the Describe Object PDU every 30 seconds for five minutes to ensure that all other simulators are aware of the object. After the five minutes Describe Object PDUs cease; however, Objects Present PDUs (another variant of `SP_PersistentObjectPDU`) are sent. Objects Present PDUs contain a list of all the objects in the current database. Simulators then compare the objects they have in their current database with those listed in the Objects Present PDU and may then request more complete information on an object in question by using an `objectRequest` variant of `SP_PersistentObjectPDU`. Upon receipt of a `objectRequest` a simulator will resend the more descriptive Describe Object PDU. This protocol is depicted in Figure 9.

The PO Protocol also establishes a method for changing persistent objects which already exist. To do this, the simulator wishing to change the object changes the `owner` field in the Describe Object PDU. The `sequenceNumber` of the object is also increased to reflect the change. The simulator which is changing ownership then transmits a Describe Object PDU with the changed owner and sequence number as well as any modifications to the object itself. The following rules are prescribed by the PO Protocol concerning the sequence numbers and owner fields of the Describe Object PDUs:

- If a simulator receives a PDU describing an object which it does not own, it will ignore the PDU if the sequence number is less than what the simulator is currently maintaining, or if the sequence number is the same and the owner is the same. If the sequence number is greater or the owner has changed it should take the information.
- If a simulator owns an object and receives a sequence number higher than is currently maintained, the simulator gives up ownership to the new simulator and updates the object.
- If a simulator owns an object and receives a PDU with an equal simulator number it compares its address magnitude with the other simulators and if higher will increment the sequence number, rebroadcast its own Describe Object PDU and maintain ownership.

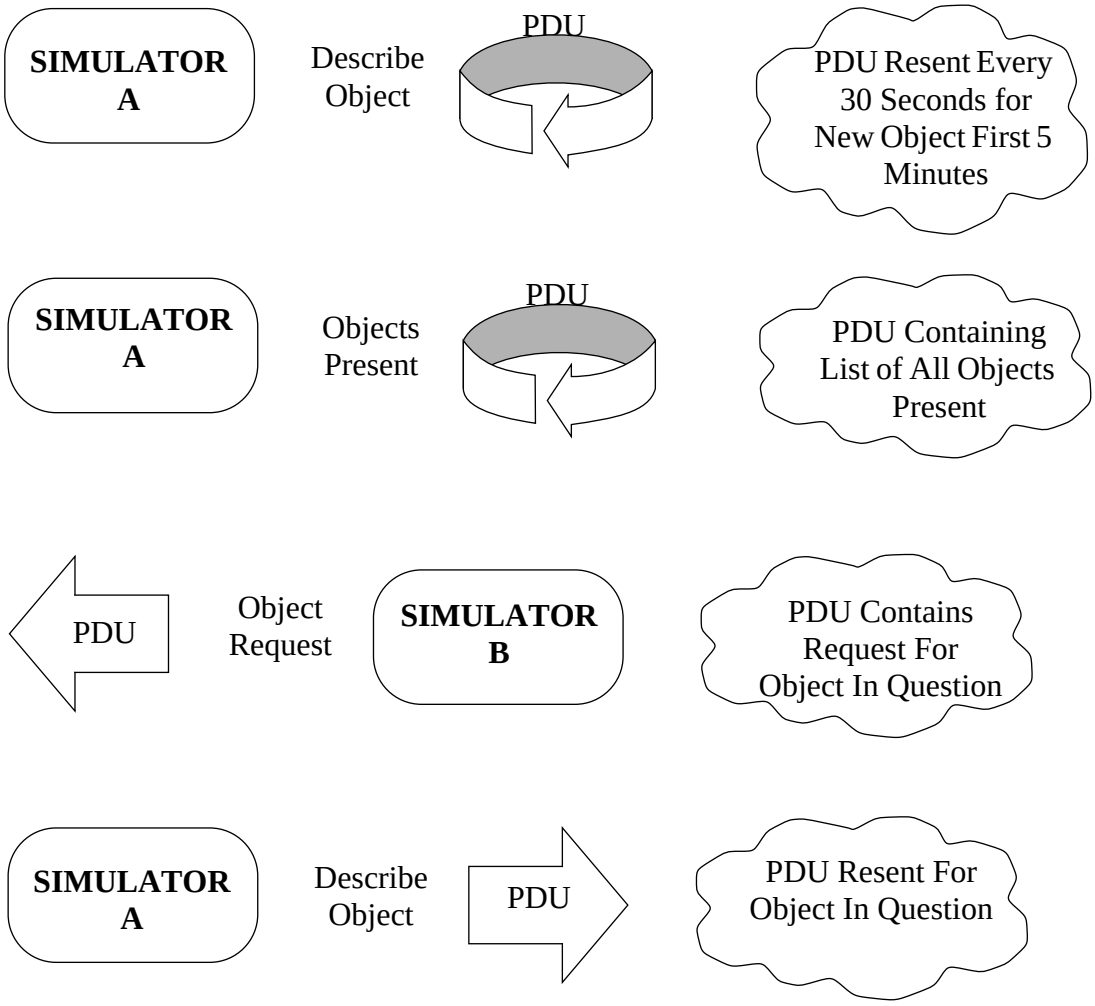


Figure 9: Creation and Modification of Persistent Object

The Sand Table uses the above rules for creating and changing persistent objects and is discussed in detail later. While the fields of the Describe Object PDU described thus far have concerned creation, ownership and updating of PDUs, the PDU contains a sub-protocol which describes the actual object in detail. `mclass` identifies what type of object the PDU actually describes. It should be noted that in the original ModSAF C library `mclass` is named `class` which caused a keyword naming conflict when integrated into the

Sand Table which was written in C++. As a programming note, the name was changed to `mclass` and a modified copy of the header file p_po.h had to be used.

The `variant` portion of the Describe Object PDU contains the various possible types of persistent objects. The variants most important to the Sand Table are those describing points, lines and minefields. Also important for reasons of ensuring that measures are displayed on the ModSAF stations is the overlay variant.

The point measure is a measure located at one position on the terrain used to describe a significant aspect of the plan at that point. An example of a point measure would be a checkpoint, which has a specific symbol describing it, a specific location on the terrain and a specific meaning with regards to creating a plan on a Sand Table. ModSAF currently supports seventeen different types of point measures which would support the vast majority of planning problems. The description of a point is contained in `SP_PointClass`, for which the structure is included in Figure 10.

```
typedef struct{
      SP_ObjectID                overlayID;
      SP_PointStyle              style;
      Sp_OverlayColor            color;
      unsigned                   dashed     :    1;
      unsigned                   _unused_8  :    31;
      SP_PointLocation           location;
      SP_Angle                   direction;
      char                       text[SP_maxPointNameLength];
}  SP_PointClass;
```

Figure 10: Describe Object Point Class Variant

The fields used to describe the point are very straightforward and will be described briefly. `overlayID` assigns the overlay in which the point measure should be displayed. As discussed, ModSAF only displays objects in overlays which are being displayed at that simulator. The Sand Table only displays a single overlay; however, this field must be used to ensure that ModSAF will display the measures created by the Sand

Table. `style` describes the particular type of point the measure represents. The seventeen possible points are contained in p_po.h. ModSAF supports five different colors also contained in p_po.h. `color` describes the particular color for the point. *Proposed* or *uncertain* measures are delineated on an overlay by representing the measures with dashed lines. This representation is possible using the `dashed` field. The `location` field gives the position on the terrain database of the point measure. `direction` enables an orientation to be included in the description of a point since some points may have an orientation with respect to the terrain onto which they are placed. Lastly, points may have text embedded within them to name a point or be more descriptive. The `text` field enables the inclusion of text in the description of a point.

Lines represent linear control measures on the Sand Table. Like points they have a unique representation, meaning and location on the database terrain. An example of a line would be a boundary line separating two units on the battlefield. Another example would be a route to be followed by a unit over the terrain. Each of the examples would be represented by a different style. The structure for line measures is given in Figure 11.

`overlayID`, `style`, `color` and `dashed` have the same properties as was discussed with the same fields in the point measures. Each line is comprised of a number of points on the terrain. The number points in each line is maintained in `pointCount`. `thickness` refers to the thickness of the line displayed on the overlay which represents the measure on the virtual terrain or map. It is the thickness of the line in pixels and concerns the display of the measure. This differs from `width` which differs by line style and represents the width of the measure on the ground expressed in meters. `beginArrowHead` and `endArrowHead` specify the style of the arrow head on the ends of the line (which may be no arrow). `closed` indicates whether the line should be represented by a closed line loop in which the last point is connected to the first point. `splined` specifies whether or not the line should be smoothed with a splining function and is not used in the Sand Table at this time. `route` indicates whether the line indicates a route or other control measures and is not used at this time. `shouldBeSimulated`, `simulated`,

```
typedef struct{
      SP_ObjectID               overlayID;
      SP_LineStyle              style;
      SP_OverlayColor           color;
      unsigned char             pointCount;
      unsigned char             thickness;
      unsigned short            width;
      SP_ArrowHeadStyle         beginArrowHead;
      SP_ArrowHeadStyle         endArrowHead;
      unsigned                  closed;
      unsigned                  dashed;
      unsigned                  splined;
      unsigned                  route;

      unsigned                  shouldBeSimulated     : 1;
      unsigned                  simulated             : 1;
      unsigned                  _unused_10            : 2;
      SP_SAFMethodology         methodology;
      unsigned long             _unused_11;
      SP_SimulationAddress      simulator;
      SP_PointDescription       points[1];
}  SP_LineClass;
```

Figure 11: Describe Object Line Class Variant

`methodology` and `simulator` are used when creating simulation representations of the lines and are not used by the Sand Table either.

The last field of `SP_LineClass`, `points`, is somewhat special and deserves closer examination. `points` is an array of points comprising the line. Points comprising the lines can be one of two types. The first is a simple x and y representation of the location of the point on the terrain. The second point representation, represents a road segment from the terrain database. The code for `SP_PointDescription`, and `SP_RoadSegment` and `SP_PointLocation` is given in Figure 12, Figure 13 and Figure 14 respectively.

The `pointType` field in the `SP_PointDescription` structure indicates how that point is represented. The variant portion contains the actual data describing the points. If the point represents a road segment the starting and ending points of the segment are given in this structure in the `startAt` and `endAt` fields. The `direction` indicates

```
typedef struct {
      short           pointNumber;
      SP_PointType    pointType;
      unsigned char   _unused_9;
      union {
            SP_RoadSegment        roadSegment;
            SP_PointLocation      location;
      } variant;
}  SP_PointDescription;
```

Figure 12: Point Description Structure

```
typedef struct {
      unsigned        index      : 24;
      SP_Direction    direction;
      short           startAt;
      short           endAt;
} SP_RoadSegment;
```

Figure 13: Road Segment Variant of Point Description

```
typedef struct{
      long x;
      long y;
} SP_PointLocation;
```

Figure 14: Point Location Variant of Point Description

whether the line should follow the segment from the first point to the last or last to first. The road segment representation of a line is very useful, especially in cases such as road marches, which proceed from one point to another only on roads. Using the ModSAF GUI overlay, the user can select lines to follow roads and subsequent lines will follow the roads of the terrain database. The Sand Table does not allow the user to automatically follow

roads at this time as does ModSAF; however, the Sand Table can receive PDUs with lines containing road segments and properly display them.

As was discussed earlier, upon initialization the Sand Table needs a terrain database for correlation and registration of points on the terrain. The Sand Table also utilizes ModSAF library libroute.a to make the conversions between road segments to points. Thus, when a line is received by the Sand Table, the point locations are either directly accessible if given in the `SP_PointLocation` form or if given in the `SP_RoadSegment` form are convertible to point locations on the terrain.

Minefields are also persistent objects which can be represented in the Sand Table. Minefields differ from point and line measures in that they are active entities managed by a simulator. Entities are the actual actors of the ModSAF simulation. The entities are objects which interact with each other such as tanks, helicopters, and infantry, differing from persistent objects which are more abstract objects. More specifically, a minefield is a measure which can have an effect on simulated units and vehicle entities in a simulation. For example, if a tank platoon drives over a minefield, casualties will be inflicted on the platoon. This is not the case with lines and points which are imaginary control measures and in fact have no physical significance in the real world. As such, minefields are much more complex measures and their PDUs have extensive capabilities to manage and control the minefield. The Sand Table uses only a small portion of the minefield capabilities at present; however, future expansion could be incorporated.

Two variants of minefields are possible with only one of the variants being used in the Sand Table. The first variant is Point Minefields and is not used. With this variant, minefields are represented by point locations of each individual mine. The other method and the one currently in use by the Sand Table is the Area Minefield. In this representation minefields are described by a perimeter and a density. The locations of the mines are represented by grids which are either on, indicating a mine is present in a grid box, or off indicating no mine is present. These grids are used in the simulation of an active minefield

and are not discussed here. The structures for SP_MinefieldClass and SP_MinefieldArea are presented in Figure 15 and Figure 16 respectively.

```
typedef struct {
      SP_ObjectID                overlayID;
      SP_LineStyle               style;
      SP_OverlayColor            color;

      char                       text[SP_maxPointNameLength];

      SP_ObjectType              munition;
      SP_ObjectType              detonator;

      SP_SimulationAddress       commander;
      SP_SimulationAddress       simulator;

      SP_MinefieldType           minefieldType;

      unsigned                   shouldBeSimulated     : 1;
      unsigned                   simulated             : 1;
      unsigned                   _unused_13            : 6;

      SP_SAFMethodology          methodology;
      unsigned char              _unused_14;

      long                       expansion1;
      long                       expansion2;
      long                       expansion3;
      long                       expansion4;

      union {
            SP_MinefieldPoints   point;
            SP_MinefieldArea     area;
      } variant;
}  SP_MinefieldClass
```

Figure 15: Describe Object Minefield Class variant

Since the Sand Table utilizes only a small portion of a minefield PDU's capabilities, only pertinent fields are discussed. overlayID, style, color and text specify the same information as was described with point and line measures. minefieldType specifies which of the two types of minefields the PDU describes. The actual minefield data is contained in the variant portion of the PDU, of which the Sand

```
typedef struct {
  unsigned short    size;
  unsigned short    rowWidth;
  unsigned short    density;
  unsigned short    pointCount;
  SP_PointLocation  perimeter[SP_maxMinefieldPerimeterPoints];
  SP_PointLocation origin;
  unsigned char     minegrid[4];
}  SP_MinefieldArea;
```

Figure 16: Area variant of Minefield

Table only implements `area`. Fields in `SP_MinefieldArea` concerning actual mines and their locations and densities are not used by the Sand Table. The fields of concern are `pointCount`, `perimeter` and `origin`. `pointCount` contains the actual number of points delineating the perimeter of the minefield. `perimeter` is an array of the actual points defining the perimeter given in x and y coordinates. Lastly, the `origin` field is the location of the southwest corner of the first minecell. While not currently implementing the specifics of the actual minecells, `origin` is used to appropriately anchor text within the minefield.

Text can be used with control measures. The PO Protocol has provisions for sending text. Figure 17 shows the text variant of the `SP_DescribeObjectVariant` PDU. `overlayID` is the overlay to which the text is assigned and is used in the same manner as it was with the other measures. `size` is the font size of the text. `length` is the length of the actual text which is contained in `text`. `location` is the coordinates of the text location. The text can also be associated with another control measure and this measure's object identification is stored in `associatedObject`. `associatedPointNumber` can be used with a line to associate the text with a specific point in the line. The remaining fields are used to correctly position the text.

```
typedef struct {
      SP_ObjectID                overlayID;
      SP_TextSize                size;
      SP_OverlayColor            color;
      short                      length;
      SP_TextAlignment           alignment;
      unsigned char              _unused_17;
      SP_PointLocation           location;
      short                      horizontalOffset;
      short                      verticalOffset;
      SP_ObjectID                associatedObject;
      short                      associatedPointNumber;
      char                       text[4];
}  SP_TextClass;
```

Figure 17: Describe Object Text Class Variant

The final persistent object used in the current Sand Table design is the overlay variant. The importance of overlays has been discussed and the actual structure of the variant of the Describe Object PDU will be briefly covered. The structure for the overlay variant is given in Figure 18.

```
typedef struct {
      char                  name[SP_maxOverlayNameLength];
      SP_OverlayColor       color;
      unsigned              scratch    : 1;
      unsigned              working    : 1;
      unsigned              _unused_6  : 6;
      SP_ForceID            forceID;
      unsigned char         _unused_7;
}  SP_overlayClass;
```

Figure 18: Describe Object Overlay Class Variant

`name` is a textual name of the overlay. The Sand Table makes use of this name by creating an overlay named “NPSNET”. This naming is used to ensure measures are displayed on both the Sand Table and the 2D GUI of ModSAF. `color` has more

significance and a slightly different meaning than similar fields in the previously described persistent objects. In the previous objects the color field simply specified the color of the measure. However, one of the options for color besides the five colors given in ModSAF is to specify that the color of a measure is the overlay default. `color` in `SP_overlayClass` is this overlay default color. When an overlay is being used the default color must be known to enable the user the option of choosing the overlay color as the color of the measure. `forceID` specifies which force should be displayed on the overlay. `SP_ForceID` is defined in basic.h of the ModSAF library. While the Sand Table does not offer an option as to what force to display, it does properly update the field to `SP_forceIDIrrelevent` which ensures that ModSAF will display *all* measures on the "NPSNET" overlay.

The PDUs and variants described are the most important segments of the ModSAF PO Protocol being used in the current design of the Sand Table. The actual Sand Table implementation of the protocol in covered in Chapter IV.

C. NPSNET OVERVIEW

While the Sand Table uses ModSAF for management of control measures and uses the PO Protocol to effectively communicate with ModSAF, the actual Sand Table system is implemented in and sits on top of NPSNET. NPSNET originated as a vehicle simulator and has evolved into a virtual world.

NPSNET is a large scale virtual environment which allows users to interact with 3D terrain, objects and players on that terrain. The system is written in AT&T C++ and follows an object oriented paradigm. The system uses a hierarchial approach when writing the functionality of vehicles and weapons. It capitalizes on inheritance of much of the functionality for vehicles from higher level classes. [ZYDA93]

NPSNET uses the Silicon Graphics, Inc. API called *Performer*. Performer handles many of the graphics specific tasks of NPSNET such as hidden surface elimination and culling [ZYDA93]. Performer uses a hierarchical directed acyclic graph (DAG) structure

to efficiently cull entire branches of a scene based on whatever scene organization was chosen by the user. The Sand Table is also written using Performer. The Sand Table is written to be superimposed over NPSNET as the ModSAF overlay is superimposed over a map of terrain. The actual code of the Sand Table is part of NPSNET; yet, the Sand Table is self-contained both in its communications and manipulation of objects. The Sand Table has only two "hooks" into NPSNET, one being functional and one being graphical. The functional one is a function call to the main Sand Table program once each time the NPSNET main function runs through its application program. The graphical hook, which can be toggled on or off during NPSNET operation, is an added branch which contains the depiction of all control measures to the NPSNET Performer scene. The Sand Table "uses" the functionality of NPSNET such as terrain and vehicle display; however, the Sand Table has no means to manipulate the terrain or vehicles.

IV. SYSTEM INTEGRATION, DESIGN AND IMPLEMENTATION

A. INTRODUCTION

Having discussed the components used to build the Sand Table, the actual design of the Sand Table and integration of the components must be considered. The overall philosophy of incorporating existing functionality into the system should be remembered. The Sand Table incorporates interaction between NPSNET and ModSAF. This interaction is accomplished by using the PO Protocol. All of the code unique to the Sand Table is written as an extension to NPSNET IV.8. An overview of the Sand Table operation will put the design into a proper context and will manifest many of the operational characteristics of the Sand Table.

1. Sand Table Operational Overview

Whereas a ModSAF station presents the user with a 2D overlay of the plan being constructed, the Sand Table could be considered a 3D overlay. The Sand Table places the user into NPSNET in the desired terrain database. The terrain database is not modified from that presented in the native NPSNET. The user can be placed in the environment as a vehicle or in a stealth mode so as not to be seen as a vehicle entity. The user can move to any part of the terrain database using native NPSNET capabilities which include keyboard, joystick or spaceball input.

Full functionality of NPSNET is retained but separated from the functionality of the Sand Table so vehicle movements, weapons firings, etc. can still be seen if there are vehicle entities on the battlefield. Similarly, the Sand Table user can behave as a vehicle and can execute any of the above actions. During a planning operation; however, it would be more logical to assume a stealth mode and view the battlefield rather than participate on it. Nonetheless, vehicle functionality is retained in the Sand Table and can be used. A scenario in which both vehicle and Sand Table functionality could both be used would be

a helicopter flying a route over the terrain. Using the Sand Table, the route to be flown could actually be visualized and followed over Virtual terrain. Whether in NPSNET as a vehicle or in stealth model all functionality of the Sand Table can be exercised. Additionally, the Sand Table can be toggled off.

In the virtual environment the user is given a Sand Table menu, the design of which will be discussed later in detail. The menu is also shown beginning with Figure 34 (see Figure 34). Using the menu, the user can select desired attributes for the control measures such as color or style and construct a control measure at a location on the terrain. The placement of the control measure is accomplished using the mouse and a 2 1/2D selection of location on the terrain. The 2 1/2D selection is simple while at the same time offering a unique solution to picking any terrain in the viewing frustum.

Once the measure is created, its existence is displayed and maintained and its attributes are sent to the network for the other Sand Tables using the ModSAF PO Protocol. ModSAF stations can also receive the PDUs from the Sand Table and can display and maintain the measures at that station. This offers the ability for workstations which do not possess the Sand Table software, yet do possess ModSAF, to participate in a planning problem albeit without a 3D terrain display.

The PO Protocol is used both for intra-Sand Table communication as well as communication with ModSAF. The Sand Table receives PDUs from other Sand Tables with attributes of control measures created at other workstations. The Sand Table will then construct the measure for display, and will maintain and update the measure. Similarly, a Sand Table can receive a PDU from a ModSAF station running an overlay for the same terrain as the Sand Table, and the Sand Table will display and maintain the measure. As depicted in Figure 19, this completes the circle of cross-system compatibility in that any Sand Table or ModSAF station can create and send a control measure and any other Sand Table or ModSAF station can receive and display control measures. Further, there is no distinct ownership of measures, so that when a measure is created by any Sand Table or

ModSAF station, any other station can move the measure on the terrain, change any of the attributes of the measure or delete the measure.

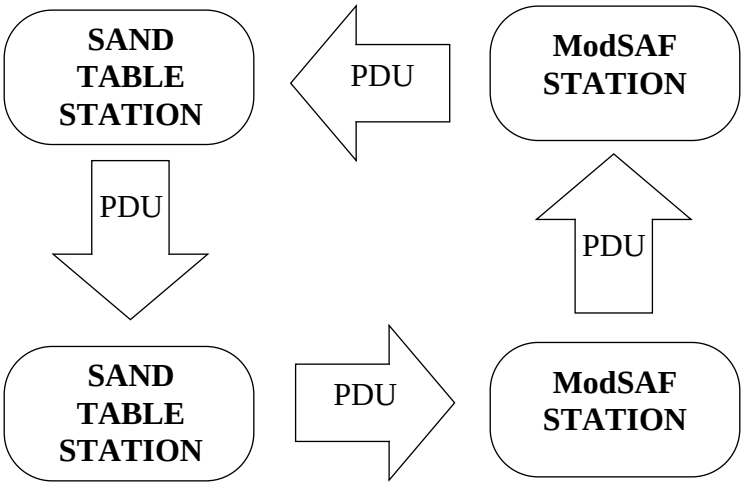


Figure 19: Possible Sending and Receiving of PO PDUs

2. SAND TABLE DESIGN OVERVIEW

In order to accomplish the operational behavior and capabilities described above, the Sand Table can be divided into several smaller elements of functionality. These are:

- Sand Table representation of control measures.
- Maintaining of known control measures.
- Processing of incoming PDU traffic.
- Processing of outgoing PDU traffic.
- Local creation of control measures created at other stations.
- Creation of control measures at Sand Table station.
- Proper display of control measures on the Sand Table.
- Manipulation and movement of control measures on the Sand Table.

These areas of functionality are by no means disjoint or mutually exclusive; however, the delineation does accentuate tasks which must be accomplished by the Sand Table. How these tasks are accomplished is covered in the following sections. A choice to use an object-oriented methodology for representation of the control measures was made. The paradigm seems a natural one since the Sand Table can be expected to maintain several different types of control measures which contain similar information yet may have different methods needed to implement needed functionality or slightly different members unique to the control measure. The PO_MEASURES_CLASS was created as the base class for control measure objects. It should be noted that while the actual control measures are implemented using an object-oriented methodology, much of the manipulation of the measures, when deemed inappropriate to be included as a member function, uses a more functional based approach.

B. SAND TABLE DESIGN AND IMPLEMENTATION

At the heart of the Sand Table design are two files, po_meas.cc and po_funcs.cc. po_meas.cc is the actual implementation of the PO_MEASURES_CLASS and contains the constructors, member functions and destructors. Logically, functions pertaining to the overall operation and state of the Sand Table are contained in po_funcs.cc which follows a more functional based paradigm. Functions associated with the actual behavior of the control measures are contained as members and member functions of PO_MEASURES_CLASS in po_meas.cc and use the object oriented approach. The separation of functionality and paradigm is not complete; however, that is the design goal. The architecture and design of the Sand Table can be understood by considering each of the elements of functionality described in the design overview and examining how the Sand Table accomplishes the task.

1. PO_MEASURES_CLASS C++ Class

Sand Table representation of control measures is accomplished using the PO_MEASURES_CLASS C++ Class. This class encapsulates a control measure's current

state as well as its functionality. Especially useful are polymorphic member functions involving display of objects as well as communications concerning objects, since the methods involved are significantly different for each different type of measure. All code concerning PO_MEASURES_CLASS is contained in po_meas.cc unless otherwise annotated. The base class and derived class code will be given here and discussed briefly. The inheritance tree for the base class and derived classes is shown in Figure 20. More detailed discussion of functionality will be given in the sections concerning the functionality which the member functions perform. The PO_MEASURES_CLASS base class is given in Figure 21.

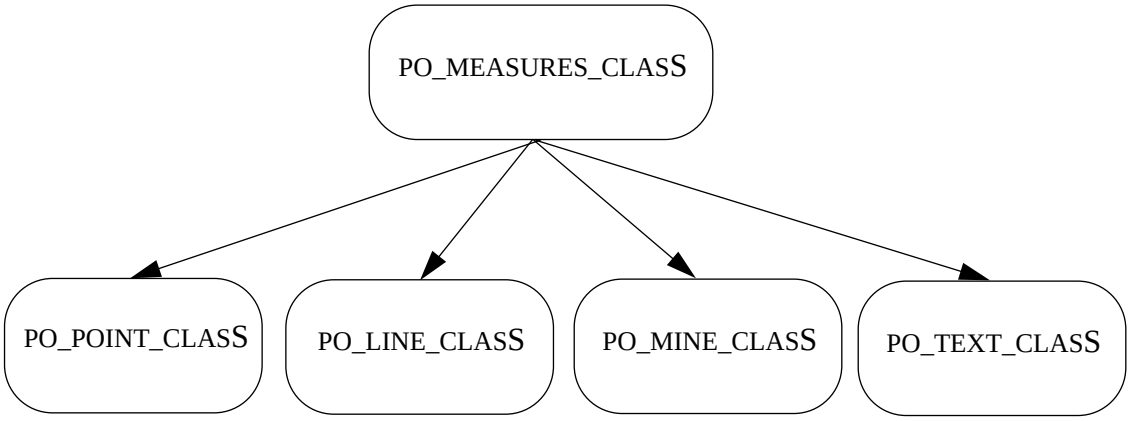


Figure 20: PO_MEASURES_CLASS Inheritance Tree

The class was designed to contain the same data as the `SP_DescribeObjectVariant` PDU which fully describes a control measure. The representation and naming may differ; however, the content is paralleled. Note, the members `dashed`, `style` and `color` correspond directly to fields in the sub-protocol of `SP_DescribeObjectVariant` for points, lines and minefields. Similarly, `numpoint` corresponds to `pointCount` in `SP_LineClass` and `SP_MinefieldClass`. `points`

```
class PO_MEASURES_CLASS {

protected:
      ushort                dashed;
      ushort                style;
      PO_COLOR              color;
      int                   numpoint;
      pfVec3                points[C2_SEGMENTS];
      pfVec3                difference;
      int                   firstTime;

public:
      PO_TYPE               type;
      pfGroup*              polys;
      char*                 pdu;

      PO_MEASURES_CLASS(SP_DescribeObjectVariant, int);
      PO_MEASURES_CLASS();
      ~PO_MEASURES_CLASS();

      virtual void update(SP_DescribeObjectVariant, int);
      virtual void movePO(float, float);
      virtual void dragUpdatePO(float, float);
      virtual void dragToNet(float, float);
      virtual void updatePDU();
      virtual void sendPDU();
      virtual pfGroup* make_meas();
      virtual void create_text_node(pfVec3, PO_COLOR, char*);
};
```

Figure 21: PO_MEASURES_CLASS C++ Class

corresponds to `location` in `SP_PointClass`, `points` in `SP_LineClass` and `perimeter` in `SP_MinefieldClass`. The specific way the members are represented and implemented are in the derived classes and will be examined later. It is important to remember how the state of a control measure is actually represented in three different ways. The first is how it is stored within the PO_MEASURES_CLASS, the second is how it is stored in the `SP_DescribeObjectVariant` PDU and the last is how the measure is actually displayed with Performer.

Additional members and member functions are also required within the class and will be discussed briefly. `difference` and `firstTime` are used when dragging and dropping

control measures over the virtual terrain. `type` actually identifies which type of control measure is being represented in an instance of the class. `polys` is the actual Performer group which contains the graphical display of the measure. Associated with each control measure is a PDU which describes the control measure using a `SP_DescribeObjectVariant`. This PDU could either be the PDU received from another station describing a control measure or if the measure was created locally, a PDU is built to describe that measure. The PDU is stored with `pdu` and its uses are described later.

The class has two constructors. The first creates an instance of the class directly from the PDU received describing the measure. Since the correlation between the fields of the PDU and the actual member values is so close the constructor consists primarily of direct assignment of values with appropriate type conversions where needed. One exception is in the construction of lines. As was covered in the previous chapter, PDUs describing lines may consist of road segments. This representation is not currently implemented in PO_MEASURES_CLASS so the conversion of road segments into points is accomplished using a ModSAF library function.

The function `update` is used when a PDU is received for an object which already exists. Its functionality is essentially the same as the constructor with the exception of creating a new object. Member data are extracted directly from the `SP_DescribeObjectVariant` PDU. From the new member data a new Performer group is created and the old one is destroyed. `movePO`, `dragUpdatePO` and `dragToNet` are used to drag and drop control measures. Basically, `movePO` is the drag, `dragUpdatePO` is the drop and `dragToNet` sends the drag to the network. Each type of control measure employs a very different method to drag and drop. A goal in the Sand Table design was to allow for effective and intuitive manipulation of control measures as well as networking the dragging and dropping. The aforementioned functions accomplish this functionality and will be described in detail later.

`updatePDU` and `sendPDU` are the methods by which an object sends its state to the network. This is another case where each object will manage its PDU differently and as such needs distinct methods for communication. These different methods are primarily the construction of `SP_DescribeObjectVariant` for the particular type or `mclass` the measure is. `make_meas` is used to actually construct the graphical representation of the control measure in Performer with great difference in the construction of a point as compared to a line or minefield. The last member function in the base class is `create_text_node` whose function is apparent from the name. The control measures can have text associated with them. `create_text_node` allows for the creation and proper placement of text within the control measures.

Each of the control measures is a derived class from PO_MEASURES_CLASS. The class definitions for PO_POINT_CLASS is given in Figure 22.

```
class PO_POINT_CLASS : public PO_MEASURES_CLASS {

protected:
      char                        text[1024];
      uint                        direction;
      PO_MEASURES_CLASS*          textnode;

      pfGroup*                    make_pt(const char*);

public:
      PO_POINT_CLASS(SP_DescribeObjectVariant, int);
      PO_POINT_CLASS(ushort, PO_COLOR, ushort, uint, char*,
                     pfVec3);
      ~PO_POINT_CLASS();
      void update(SP_DescribeObjectVariant, int);
      void movePO(float, float);
      void dragUpdatePO(float, float);
      void dragToNet(float, float);
      void updatePDU();
      void sendPDU();

      pfGroup* make_meas();
}
```

Figure 22: Derived Class PO_POINT_CLASS

The PO_POINT_CLASS has additional members to describe the point. The first is `text` and `direction` which correspond directly to the fields in the `SP_PointClass` variant of the `SP_DescribeObjectVariant`. `textnode` is a pointer to additional text which may be associated with the point. `make_pt` is an additional function which is used by `make_meas` to create the Performer representation of the point.

The class has an additional parameterized constructor. This is used for the creation of a new PO_POINT_CLASS object from the Sand Table. The parameters are used to pass style, color, dashed, direction, text and the location of the point. The first constructor with `SP_DescribeObjectVariant` as a parameter would be used for the creation of a point after receiving a PDU from the network.

The next derived class is for line measures. The class definition for PO_LINE_CLASS is given in Figure 23.

Similar to the PO_POINT_CLASS, the PO_LINE_CLASS has the additional members which are fields in the `SP_DescribeObjectVariant` PDU to describe a line which include `thickness`, `closed`, `splined`, `width`, `beginArrowHead` and `endArrowHead`. `make_meas` creates the Performer graphical representation of the measure. `makeQuickLine` is used when dragging lines over terrain. The function provides the Sand Table with a low level of detail Performer representation of the line so it can be moved in real-time.

PO_LINE_CLASS also has two constructors. The first builds the object from a PDU received off the network and the second is for local creation of an object on the Sand Table. The parameters in the second constructor are the member values assigned to determine the state of the control measure. They include dashed, style, color, thickness, closed, splined, width, arrow head placement and a pointer to the point locations.

One additional parameter is included and is named `quick` in the actual code body. This is used to create an instance of a PO_LINE_CLASS measure *without* creating the Performer graphical representation. This is used when creating a line locally. Creating a full level of detail line is quite time consuming. This parameter serves as a switch to

```
class PO_LINE_CLASS : public PO_MEASURES_CLASS {

protected:
      ushort          thickness,
                      closed,
                      splined,
                      width,
                      beginArrowHead,
                      endArrowHead;
      pfGeode*        makeQuickLine();

public:
      PO_LINE_CLASS(SP_DescribeObjectVariant, int);
      PO_LINE_CLASS(ushort, ushort, PO_COLOR, ushort, ushort,
                    ushort, ushort, ushort, ushort, int,
                    pfVec3*, ushort);
      ~PO_LINE_CLASS();
      void update(SP_DescribeObjectVariant, int);
      void movePO(float, float);
      void dragUpdatePO(float, float);
      void dragToNet(float, float);
      void updatePDU();
      void sendPDU();

      pfGroup* make_meas();
      pfGeode* make_vertical();
}
```

Figure 23: Derived Class PO_LINE_CLASS

prevent the actual pfGroup from being made and is used when building lines on the actual terrain.

The last unique member of the PO_LINE_CLASS is the function `make_vertical`. This function is used when "picking" lines with the mouse. Its exact purpose and functionality will be discussed in the next chapter.

The next derived class is for minefields. The declaration for this class is given in Figure 24.

The first seven members of the class again correspond directly to the `SP_DescribeObjectVariant` PDU fields and represent the same data concerning the state of the minefield. Recalling (see page 38) that a minefield is an actual simulated entity,

```
class PO_MINE_CLASS : public PO_MEASURES_CLASS {

protected:
      ushort                minefieldType;
      uint                  munition;
      ushort                size,
                            rowWidth,
                            density;
      pfVec3                origin;
      char                  text[1024];
      PO_MEASURES_CLASS*    textnode;
      pfGeode*              makeQuickMine();

public:
      PO_MINE_CLASS(SP_DescribeObjectVariant, int);
      ~PO_MINE_CLASS();
      void update(SP_DescribeObjectVariant, int);
      void movePO(float, float);
      void dragUpdatePO(float, float);
      void updatePDU();
      void sendPDU();

      pfGroup* make_meas();
}
```

Figure 24: Derived Class PO_MINE_CLASS

some of the members such as `munition` are not currently used in the Sand Table. `textnode` is a pointer to text which may be associated with the minefield and `makeQuickMine` has similar functionality to `makeQuickLine` in creating a low level of detail representation of the measure.

Similar to the constructor for line and point measures, PO_MINE_CLASS has a constructor for both creation using a PDU or values created locally. The additional member functions are similar in function to the functions of the other derived classes and will be described in detail later.

The final derived class is PO_TEXT_CLASS and is provided in Figure 25. Text can be found by itself on the Sand Table or with other control measures giving amplifying information. Like the other control measures, the members of PO_TEXT_CLASS closely

```
class PO_TEXT_CLASS : public PO_MEASURES_CLASS {

protected:
      SP_TextSize                size;
      short                      length;
      char                       alignment;
      SP_PointLocation           location;
      short                      horizontalOffset;
      short                      verticalOffset;
      SP_ObjectID                associatedObject;
      short                      associatedPointNumber;
      char                       text[1024];

public:
      PO_TEXT_CLASS(SP_DescribeObjectVariant, int);
      ~PO_TEXT_CLASS();
      void update(SP_DescribeObjectVariant, int);
      void create_text_node(pfVec3, PO_COLOR, char*);
      pfGroup* make_meas();
};
```

Figure 25: Derived Class PO_TEXT_CLASS

reflect the `SP_DescribeObjectVariant` PDU. The class contains only one constructor which takes the `SP_DescribeObjectVariant` PDU as a parameter. In the current version of the Sand Table, text can only be received from a ModSAF station. Text can either be received as an object or it can be embedded in an object as was seen with points. `create_text_node` and `make_meas` actually construct and place the text.

2. Maintaining of Known Control Measures

By its title, this task seems to conflict with the goal in the design of the Sand Table to utilize ModSAF to maintain control measures. However, in actuality, the task does not conflict. The Sand Table must maintain a local database which represents the current control measures on the Sand Table for its own use. This is needed for the Sand Table to properly display the measures and be able to modify the measures. Where the Sand Table does *not* manage the objects is in the repeated sending of `SP_DescribeObjectVariant` PDUs upon object creation, the sending of subsequent `SP_ObjectsPresentVariant`

PDUs which list the objects in the database nor the rebroadcast of `SP_DescribeObjectVariant` PDUs when other simulators send requests for object descriptions. (see Figure 9)

po_funcs.cc contains functions which maintain the overall state of the Sand Table. Within po_funcs.cc the database of objects is stored as an array which contains all of the control measures which the Sand Table has current information on and is currently displaying. To be in the array, either the measure was created locally on the Sand Table or a `SP_DescribeObjectVariant` PDU was received from another simulator.

The array containing the database consists of objects of class `poLookUp` which consist of a key containing an object identifier of type `SP_ObjectID` and a pointer to an object of class PO_MEASURES_CLASS. When new measures are added, their keys and pointers are added at the end of the array. When deleted, they are removed from the array and the array is then compressed to fill the hole. The simple array structure was determined to be a sufficient data structure to contain the control measures since the number of expected control measures in a Sand Table problem would be relatively low. If key lookup proved to be a significant slow-down, the `poLookUp` objects could be stored in a more sophisticated data structure.

3. Processing of Incoming PDU Traffic

Within po_funcs.cc is the function `po_net_read`. This function reads PDUs of type `SP_PersistentObjectPDU` from the network and, after checking that the PDU has the proper exercise and database identifiers, calls the appropriate function within po_funcs.cc to process the PDU. po_funcs.cc currently processes `SP_PersistentObjectPDU` variants of type `SP_SimulatorPresentVariant`, `SP_DeleteObjectsVariant` and `SP_DescribeObjectVariant` with other variants being disregarded at this time. The processing of incoming PDUs is shown in Figure 26. All processing functions referred to below are contained in po_funcs.cc.

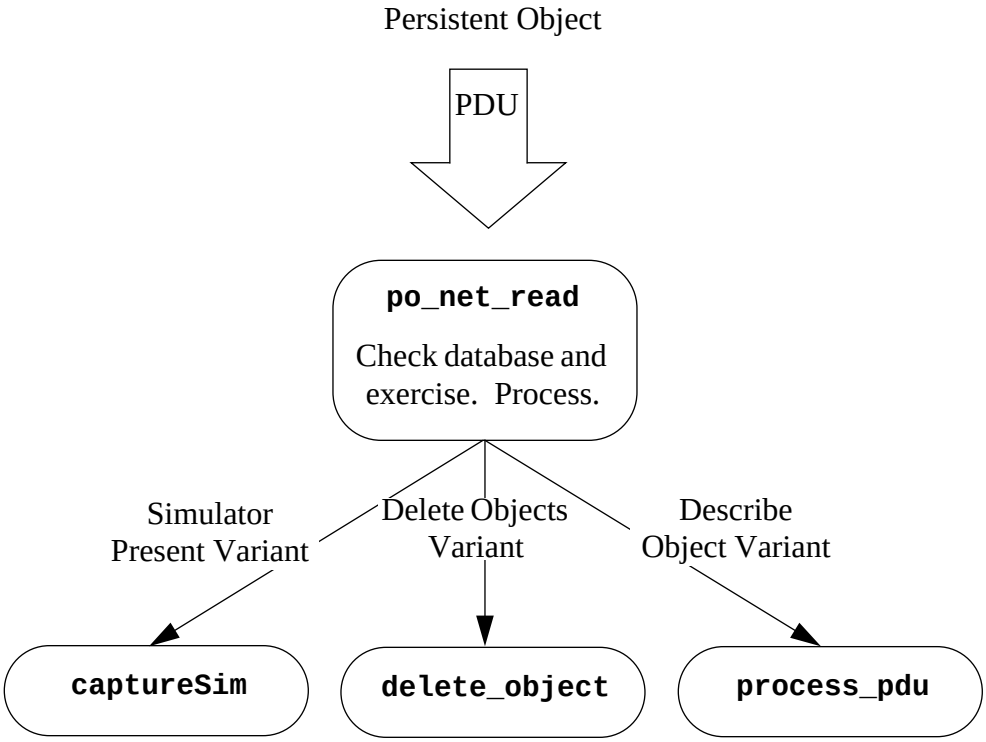


Figure 26: Processing of Incoming PDU Traffic

Upon receipt of `SP_SimulatorPresentVariant` PDUs, the Sand Table will capture a "slave simulator" if a capture has not already occurred. The function which accomplishes the capture is named `captureSim`. The purpose of capturing a simulator is to facilitate using ModSAF stations to manage control measures. The `SP_SimulatorPresentVariant` PDU provides the simulator site, host and simulator type. The type is checked to ensure that the PDU is not from another Sand Table (see page 27) and if not, objects created at the Sand Table will be passed to the "slave simulator." This is shown in Figure 27.

Upon receipt of `SP_DeleteObjectsVariant` PDUs the Sand Table compares the list of objects to be deleted as specified in the PDU with the Sand Table's database of objects. The deletion function within the Sand Table is named `delete_object`. The

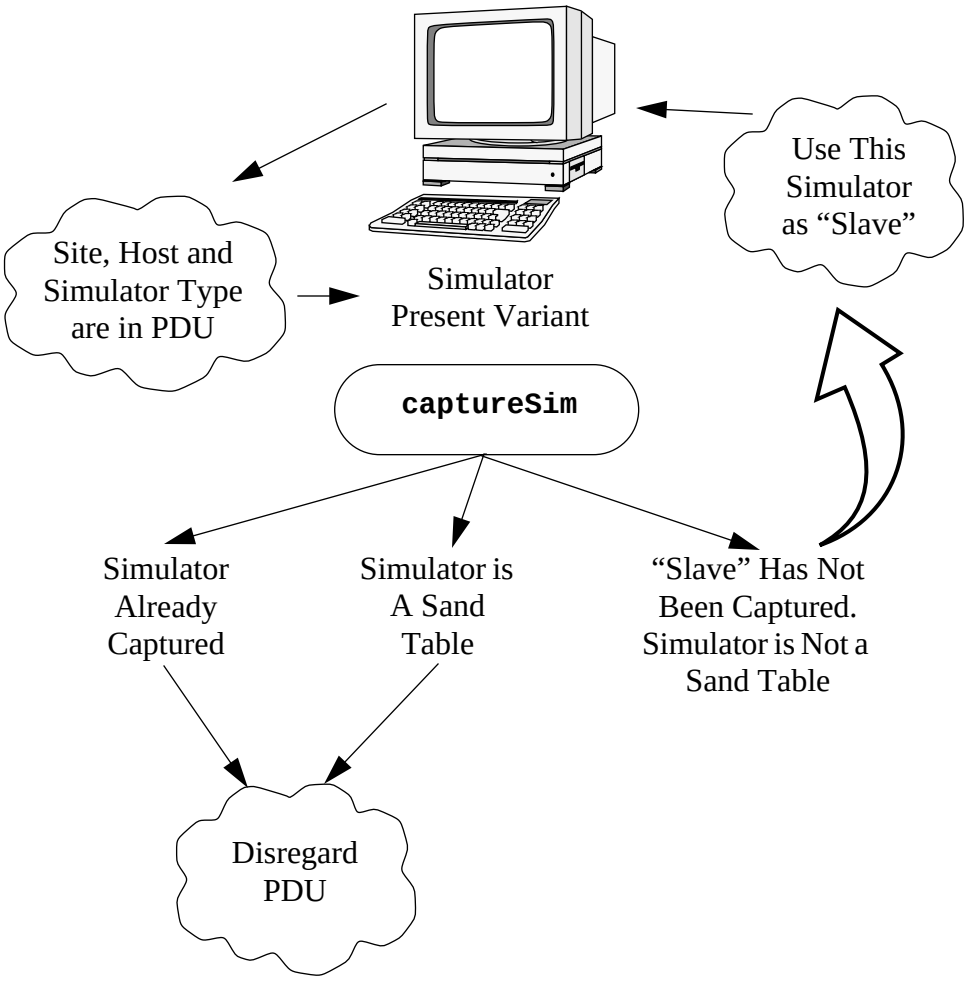


Figure 27: Processing of Simulator Present Variant

function uses the `objectID` from the array of `SP_ObjectIDWorldStateIDPair` in the PDU as a key in locating the object within the object database of the Sand Table. This is shown in Figure 28.

The receipt of a `SP_DescribeObjectVariant` calls the function `process_pdu` which is slightly more complicated but still straightforward. As pointed out in the previous chapter, the `SP_DescribeObjectVariant` PDU can be used for both creation and

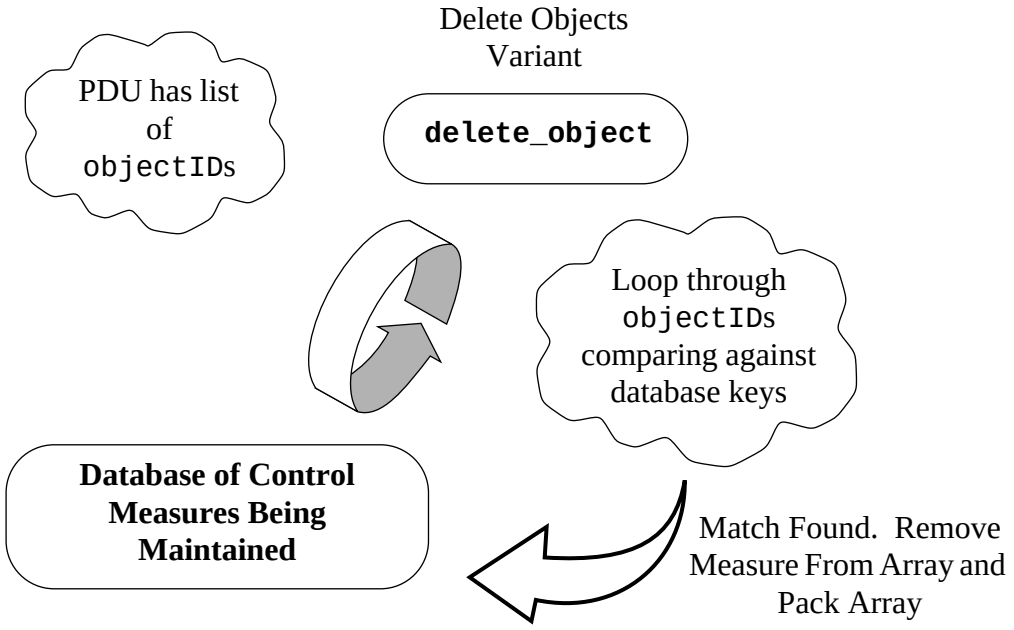


Figure 28: Processing of Delete Objects PDU

modification of control measures. Further, when an object is created the simulator owning the measure will, in accordance with the PO Protocol, broadcast `SP_DescribeObjectVariant` PDUs every 30 seconds for five minutes. Thus, the PDU received by the Sand Table could be a new object, a modified old object or a duplicate `SP_DescribeObjectVariant` PDU of an old object.

Within `process_pdu` the `objectID` of the control measure being described in the PDU is searched for within the object database of the Sand Table. If found, the PDU describes an old object. Included as a member within the PO_MEASURES_CLASS is a string representation of the PDU which created or last updated the measure. The string representation of the old PDU is then compared with the incoming PDU. If they match, the incoming PDU is a duplicate and is ignored. If different, a member function of the PO_MEASURES_CLASS, `update`, modifies the control measures attributes and changes

its display if necessary. The function `update` takes the actual PDU as a parameter to update the object's members. The function exhibits polymorphic behavior and is used with all types of control measures.

If the object contained in the `SP_DescribeObjectVariant` PDU is not found in the local database the Sand Table adds the measure. The sub-protocol of the PDU is examined to determine the type or `mclass` of the control measure and the appropriate constructor is used to create an object of type PO_MEASURES_CLASS. A pointer to the object and the `SP_ObjectID` are then placed in the array which contains the database of the current control measures known by the Sand Table. The possible flows of the `SP_DescribeObjectVariant` PDU are shown in Figure 29.

4. Processing of Outgoing PDU Traffic

Due to the limited management nature of the Sand Table, outgoing traffic is equally limited and concise. Outgoing PDUs from the Sand Table can be put into two categories, the first being "handshaking" with ModSAF and the second being PDUs concerning objects. The first category is considered to be the responsibility of the entire Sand Table and as such, the functions participating are contained in po_funcs.cc which controls the state of the Sand Table. The second category of functions are specific to the actual objects and as such are contained as member functions within PO_MEASURES_CLASS.

a. Handshaking

As soon as a Sand Table initially opens its network connection, the Sand Table takes two actions to begin "handshaking" with ModSAF. The first action is to send a `SP_SimulatorPresentVariant` PDU. The purpose is to alert a ModSAF station to the Sand Table's presence so that the Sand Table can learn of the objects currently present in the database. Recalling the PO Protocol, when a new simulator is detected the owners of persistent objects begin transmitting `SP_DescribeObjectVariant` PDUs for all owned objects so new simulators can learn the objects present (see Figure 6). After sending the `SP_SimulatorPresentVariant` PDU, the Sand Table then processes the

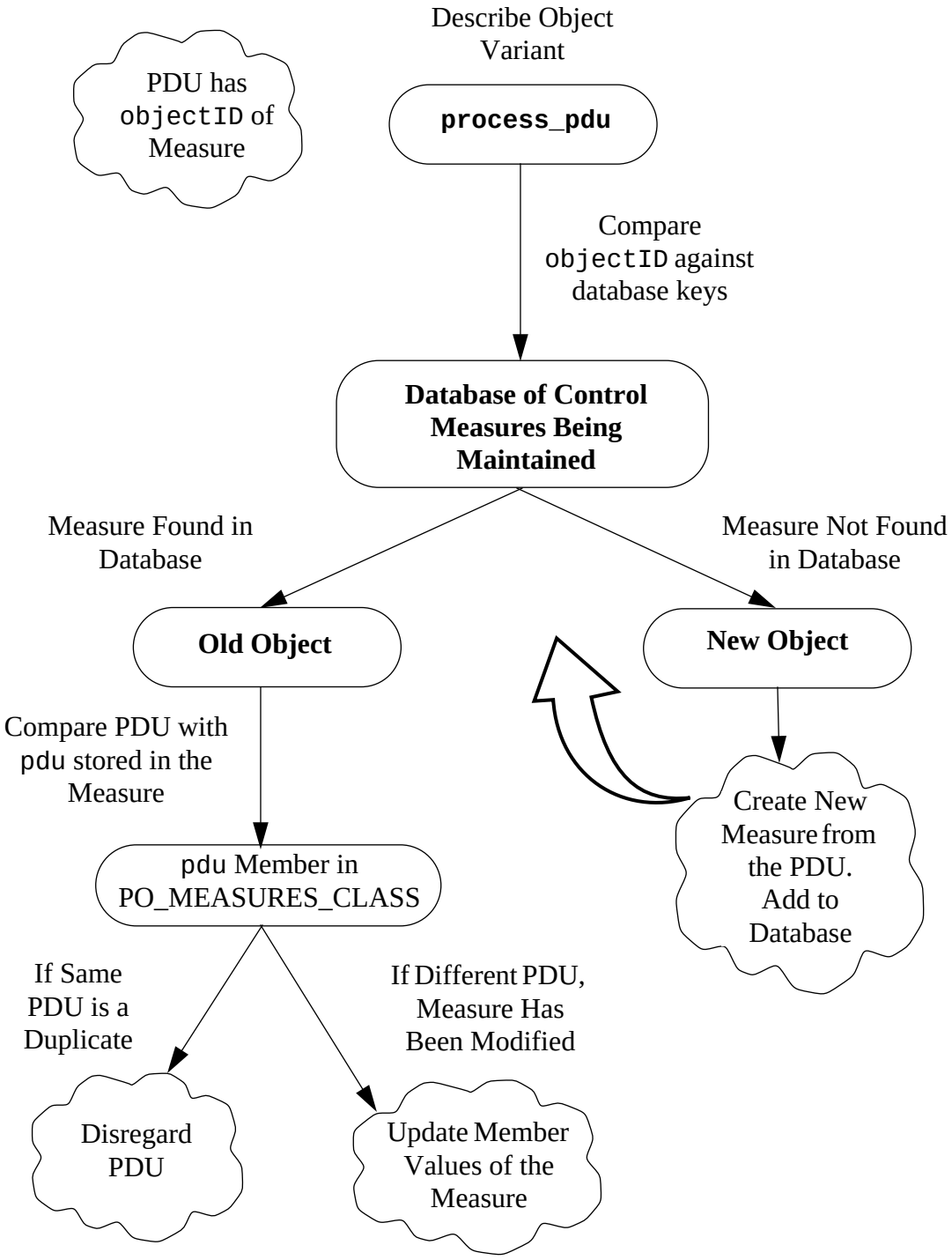


Figure 29: Processing of Describe Object PDU

SP_DescribeObjectVariant PDUs from all ModSAF stations present and builds a local database as described above for display and manipulation of the objects.

The function sendSimPresent initiates the above actions by creating a SP_PersistentObjectPDU (see Figure 4) of the simulator present variant (see Figure 5). A default protocol version is supplied to fill the version field of the PDU. Database and exercise identifiers are set to user settings on start-up and are used to fill the appropriate PDU fields. The Sand Table fills the simulator field of the PDU with its host's host and site numbers. simulatorType is set to unknown and the databaseSequenceNumber is initialized to zero. All other fields are initialized to zero and the PDU is sent.

The second "handshaking" action taken is to ensure that the Sand Tables and any ModSAF stations are displaying the overlay named "NPSNET". The action is primarily for the ModSAF stations which only display overlays selected by the user. Any object created must include the overlay it is assigned to in order to be displayed. The Sand Table is designed to only display the "NPSNET" overlay, nonetheless, it must be coordinated with ModSAF.

Initially, the Sand Table was designed to listen to the network for a predetermined time to determine if an "NPSNET" overlay was already present. This was done by examining SP_DescribeObjectVariant PDUs of the overlay variant. If an "NPSNET" overlay was detected, it was captured and used in subsequent SP_DescribeObjectVariant PDUs. If, however, an overlay was not captured, the Sand Table would create an overlay. The Sand Table would use its own site and host in the objectID field of the SP_DescribeObjectVariant PDU; however, the owner field would be filled with one of two possibilities.

If a "slave" simulator had been captured prior to building the overlay, it would be given ownership of the overlay. If no simulator capture had occurred, the Sand Table would assume ownership. The Sand Table would then continue to try to capture a simulator and as soon as it did, the Sand Table would pass ownership of the overlay to the "slave" simulator. This method is shown in Figure 30.

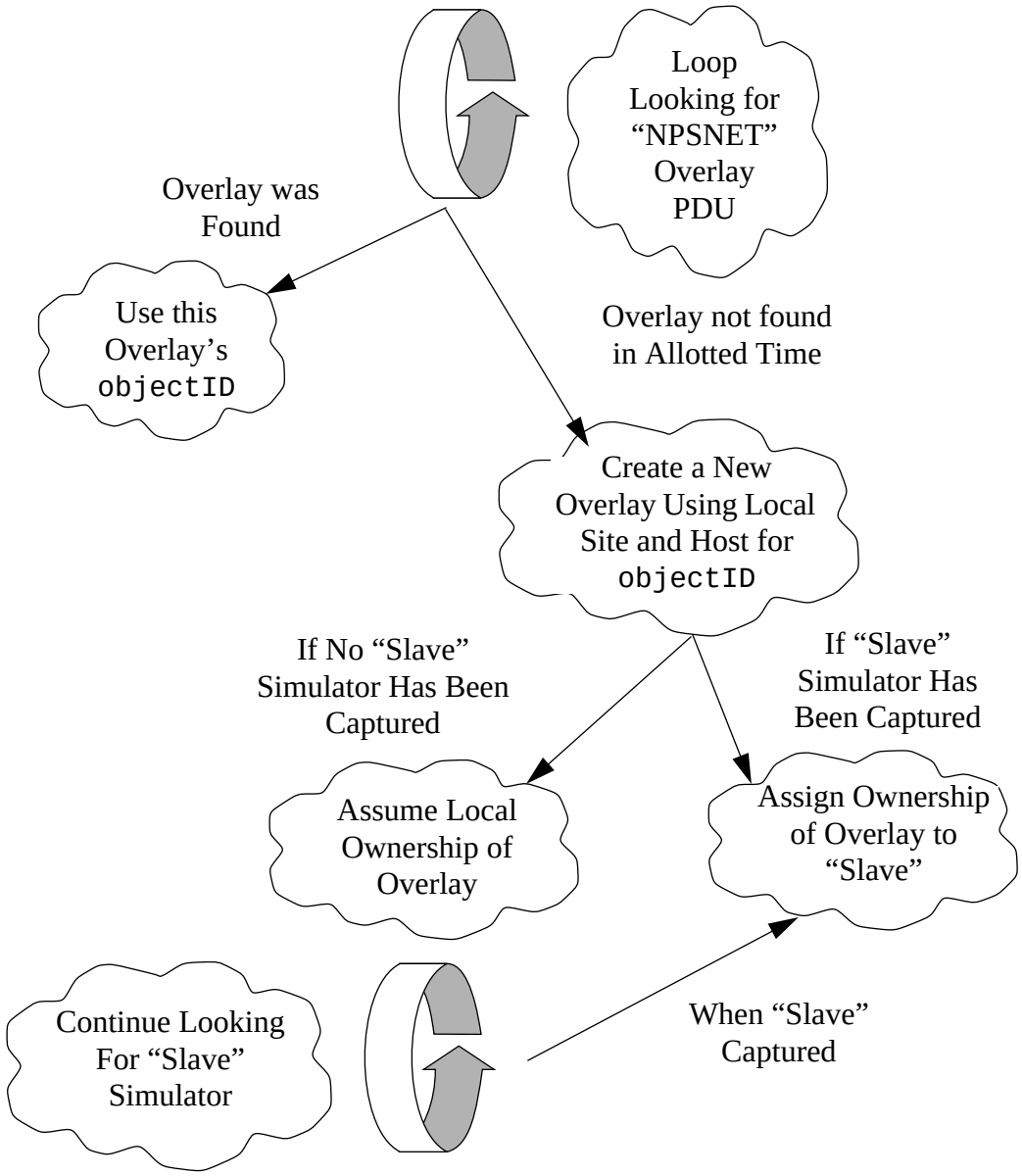


Figure 30: Incorrect “Handshaking” Protocol

This method of handshaking, while seeming initially correct, was determined to be unacceptable due to the existence of timing dependencies. A race condition existed in

the order and timing in which simulators were brought on line.

If two Sand Tables were brought up at the same time, it was possible that neither would find an overlay present at which time each would create its own overlay using their own site and host information for overlay initialization. As soon as a "slave" simulator was captured, ownership of *both* "NPSNET" overlays would be passed. This created an ambiguity in that two (or more) *different* overlays named "NPSNET" would exist. If this case existed, each Sand Table would be using its own exclusive overlay. Further, if objects were created in ModSAF, they would potentially have to be assigned to the multiple ambiguous "NPSNET" overlays. This condition is shown in Figure 31.

The second method for overlay "handshaking" proved to be successful and actually simpler. With this method an overlay is created immediately upon start up by all Sand Table stations. However, the *same* overlay is created. In order to accomplish this the `objectID` of the overlay is filled with a dummy site, host and object number, which is the *same* for all Sand Table stations. The `SP_DescribeObjectVariant` PDU is sent to the net on start up. This creates the possibility of many duplicates of the same overlay on the network; however, this is no problem since the PO Protocol already uses and handles duplication of `SP_DescribeObjectVariant` PDUs. The final issue in this method is ownership of the overlay and this is dealt with in a manner similar to that used in the original flawed overlay method.

Upon creation of the overlay with dummy defaults ownership must be determined. If the Sand Table has captured a "slave" simulator ownership is assigned to that "slave." The revised overlay "handshake" is shown in Figure 32. The race condition situation with the revised protocol is shown in Figure 33.

If, however, a capture has not occurred, the Sand Table assumes ownership and will rebroadcast the `SP_DescribeObjectVariant` PDUs describing the overlay. Upon capture of a simulator, ownership will be transferred to the ModSAF station and the Sand Table will no longer rebroadcast PDUs. If, by chance, two Sand Tables capture different

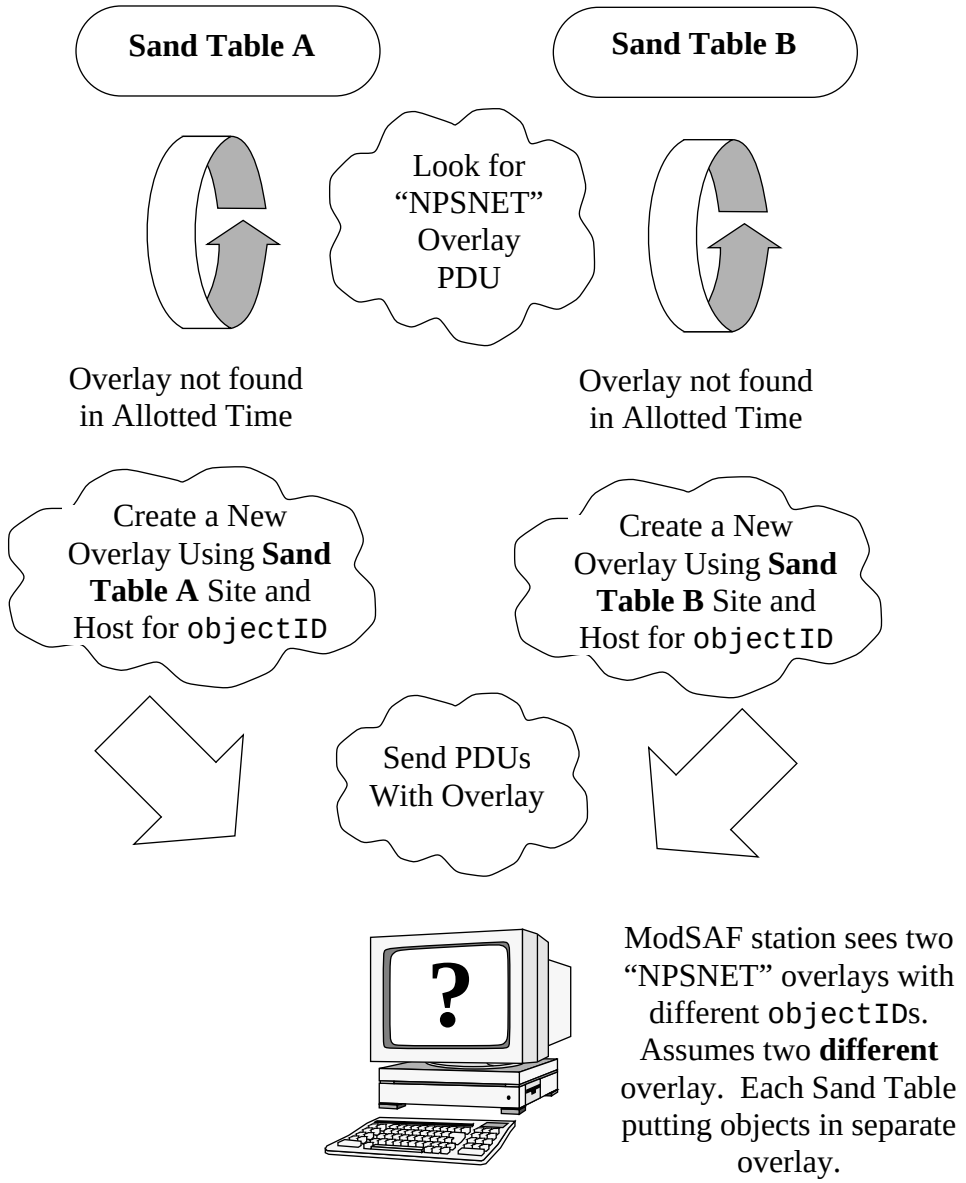


Figure 31: Overlay Race Condition

"slave" simulators, ownership will be sorted out by the PO Protocol as discussed in the previous chapter (see page 32).

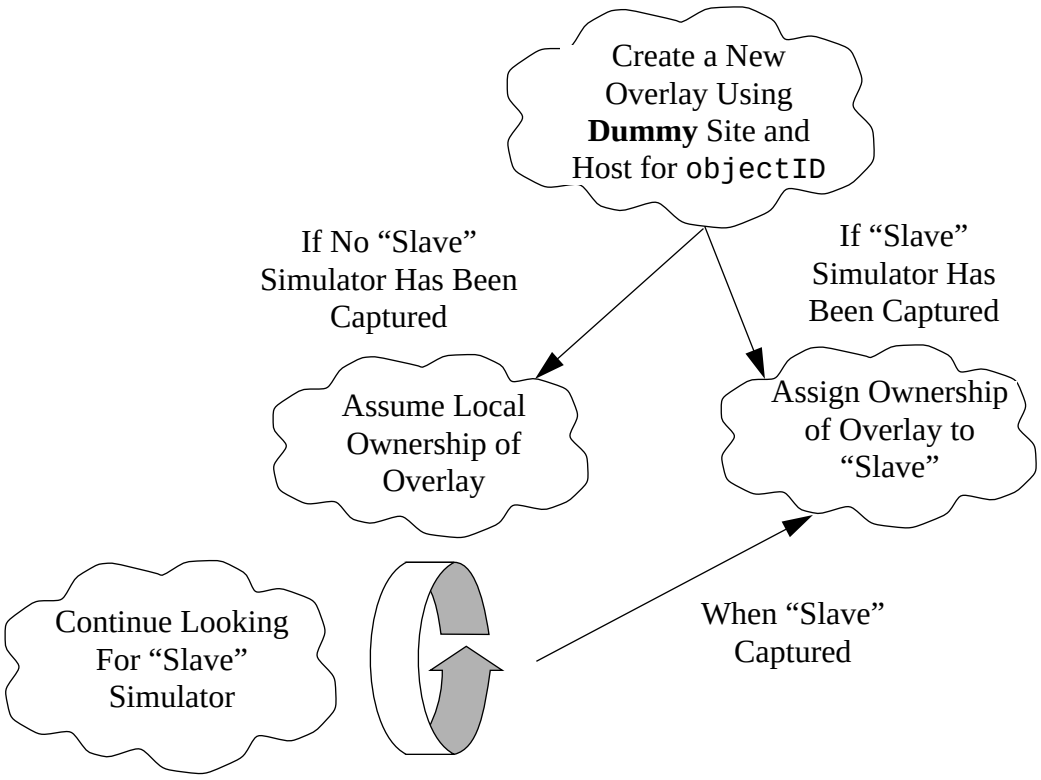


Figure 32: Correct "Handshaking" Protocol

b. Object messages

Once handshaking has been completed the Sand Table is able to send PDUs pertaining to the actual objects in the planning problem. This is one of the most important aspects of the Sand Table in that it enables the Sand Table to transmit messages concerning the creation, placement and manipulation of objects which is a major motivation for creating the Sand Table. Since the control measures are of different types it made sense for communications relating to an object to be a member function within the class of that object.

All communications sent by objects utilize PDUs of type `SP_PersistentObjectPDU`. Within each derived class of PO_MEASURES_CLASS

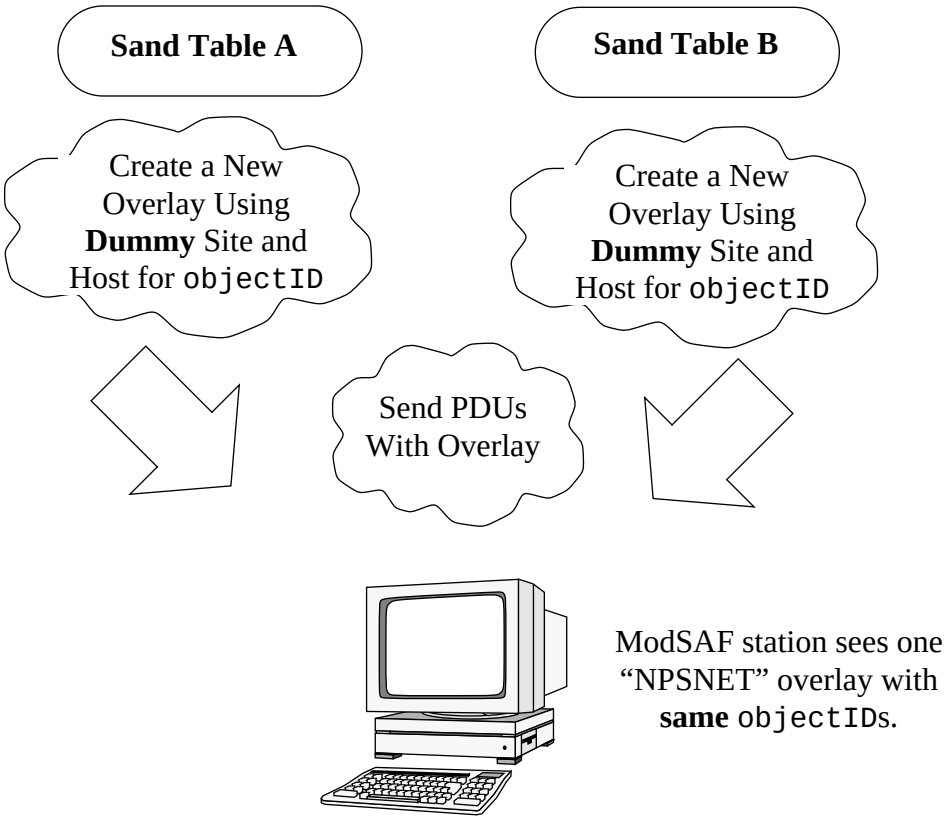


Figure 33: Corrected Race Condition

are two functions, `updatePDU` and `sendPDU`, which communicate the current state of an object to the network via a `SP_PersistentObjectPDU`. The first function, `updatePDU`, loads the member values describing the object into a `SP_DescribeObjectVariant`. The second function, `sendPDU`, completes the job by building the rest of the `SP_PersistentObjectPDU` and putting the PDU on the network.

`updatePDU` can both update the stored PDU for an object or it can actually create a PDU for a new object. The function first checks to see if the member `pdu` is NULL. Recall (see page 50), `pdu` stores the most recent `SP_DescribeObjectVariant` PDU for the object. If `pdu` is in fact NULL, there exists no PDU describing the object. This could

occur from the local creation of an object which needs a PDU to be built for it. If this is the case, storage for `SP_DescribeObjectVariant` is allocated and the header portion of the PDU is filled. `databaseSequenceNumber` is set to zero, and the `objectID` is set using the host and site of the Sand Tables workstation. A number unique to that workstation is assigned as the object number portion of the `objectID` field. The owner is set to the site and host of the "slave" simulator if one has been captured. `sequenceNumber` is set to one, `mclass` is set corresponding to the actual type of object for which the PDU is being built and in the header `overlayID` is set to the identifier of the "NPSNET" overlay. After these portions of the PDU have been created the pointer to the new `SP_DescribeObjectVariant` is assigned to the member `pdu`.

`updatePDU` could also find that the member `pdu` is not NULL, meaning that a PDU describing the object is already present. In this case a new `SP_DescribeObjectVariant` need not be created; rather, the current `pdu` will be modified. After checking `pdu`, the functions then increments `sequenceNumber`. Recalling the PO Protocol (see page 32), this increment will force the other simulators to accept the `SP_DescribeObjectVariant`. Lastly, the actual member values of the object, such as color, style and dashed, are put into the corresponding fields of the PDU. The constructor already showed this correspondence and filling the PDU is a straightforward assignment of values with exact assignment depending on the variant of the sub-protocol. It should be noted that the actual member values were created or modified elsewhere and the PDU is loaded with the current values of the members. Upon exit from the function the member `pdu` has the most recent PDU describing the object.

Recalling the `SP_PersistentObjectPDU` structure (see Figure 4), the `SP_DescribeObjectVariant` is a variant within the whole PDU. So while `updatePDU` creates the correct variant portion, the rest of the persistent object PDU must be constructed. `sendPDU` finishes this packaging and sends the PDU on the network. `sendPDU` creates a new `SP_PersistentObjectPDU`. Header information such as `version`, `exercise` and `database` are set with values which are constant throughout

the running of a Sand Table. `kind` is set to `SP_describeObjectPDUKind`. Next, the variant portion of the PDU is set. This is accomplished by loading the contents of the member `pdu` into the variant portion of the `SP_DescribeObjectVariant` PDU. Lastly, the length of the PDU is set and the PDU is sent to the network. Normally, when changing an object, first `updatePDU` would be called to make `pdu` current, then `sendPDU` would be called to send the PDU. However, if in the future the Sand Table managed more of its objects, `sendPDU` could be used at any time to send the current state of an object.

5. Local Creation of Control Measures Created at Other Stations

Much of the local creation of control measures created at other stations was covered in the discussions of the PO_MEASURES_CLASS, local object maintenance and receiving incoming PDUs. However, now that each of the components in the creation of objects has been discussed an overall view of the process can be seen. Upon receipt of a PDU, which has been determined not to currently exist in the local database, a new object of a derived class of PO_MEASURES_CLASS is created. The key to the creation is close correlation between the contents of the PDU and the actual members within the class of the object being created. With this correlation the constructor uses the actual `SP_DescribeObjectVariant` portion of the PDU and extracts the object's state information directly. After creation using the constructor, the object is put into an array containing keys and pointers to PO_MEASURES_CLASS objects. Additionally, the graphical representation of the measure is added to the NPSNET scene graph. Subsequent manipulation, changes or deletions are handled polymorphically.

6. Creation of Control Measures at Sand Table Station

The creation of objects on the Sand Table is accomplished using a Cone Tree menu system. The Cone Tree system will be described here only as it relates to the creation of control measures on the Sand Table with a detailed examination of the menu implementation being covered in Chapter V. The Cone Tree menus are 3D trees which are

oriented vertically and can be placed on the terrain. At the end of each branch an icon is present representing a menu choice. Figure 34 shows a cone tree with base level icons only.

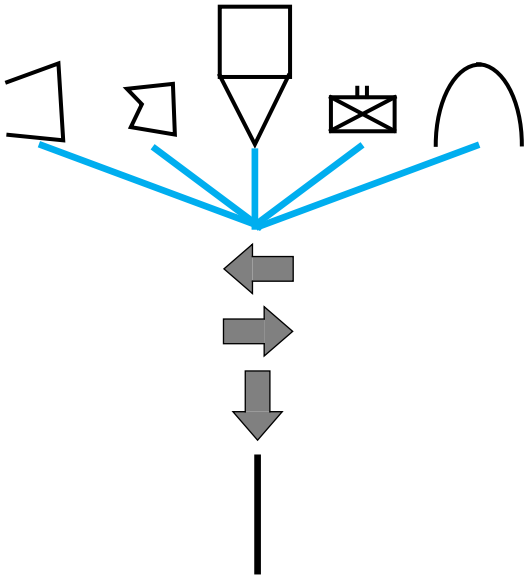


Figure 34: Cone Tree Menu

Selection of one of the icons on a tree branch will expand the tree upward by one level. When an icon is selected, the icon, along with the branch leading to the icon, are highlighted giving an indication as to what icon has been selected. This can be seen in Figure 35, where a general point has been selected and the menu has been expanded by one level.

The expansion will proceed by what action the user is trying to accomplish, for example building a point. The user will continue selecting icons up the tree until a terminator has been reached. Figure 36 shows a check mark at the top of the tree expansion which is a possible terminator. Once a terminator has been reached the user will select it and the choices the user made in expanding the tree will be implemented.

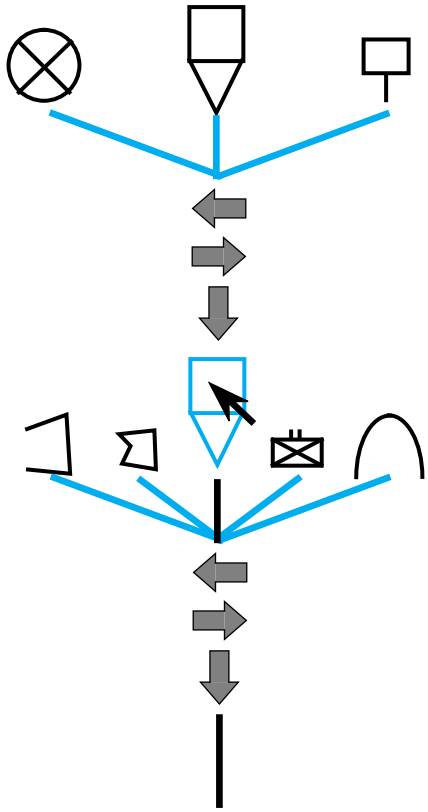


Figure 35: Menu Expanded One Level

Creation of objects is directly linked with the Cone Tree menu system. The system currently allows for the creation of points, open lines and closed lines. Icons for each of these measures can be seen in Figure 34 with the point icon being the military symbol for a general or check point (square atop a triangle), the open line being the line strip (backwards deformed letter “C”) and the closed line loop being the irregular shaped polygon. How objects are created can best be explained by following examples of creation with the menu.

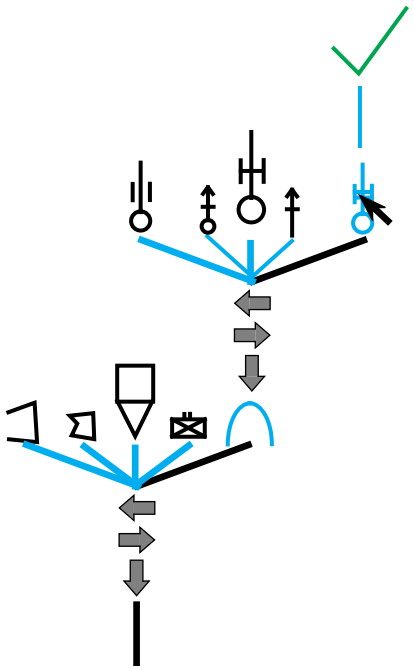


Figure 36: Expansion of Menu to Terminator

a. Point Creation

To create a point the user would select the general point at the lowest level of the menu tree. The menu would then expand offering three choices which represent the point style. This was shown in Figure 35. The next step is to select one of the point measures to be built. Figure 37 shows the selection of a coordinating point (circle with a "X" in it) and the next level of expansion.

At this level of expansion the user is presented with several circles each a different color. Selection of one of the color panels expands the menu to a terminator which is shown in Figure 38.

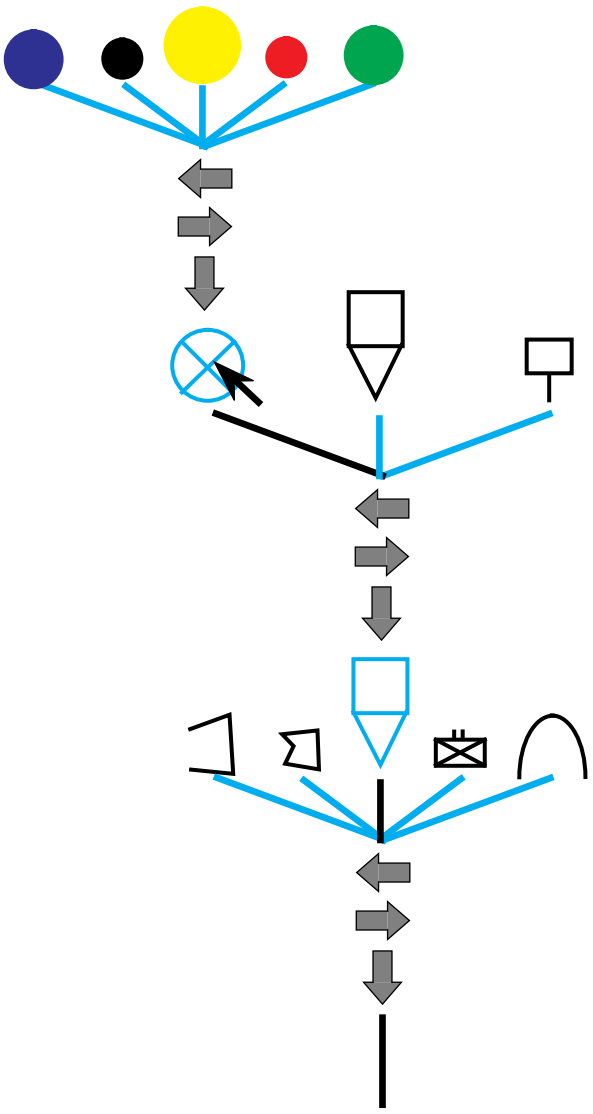


Figure 37: Second Expansion Building a Point

The final selection of the terminator then places a point of the selected type and selected color at the point where the menu base intersects the terrain. The menu then collapses and disappears. Having outlined the creation of an object from the graphical standpoint, the object creation and network aspects can be more easily understood.

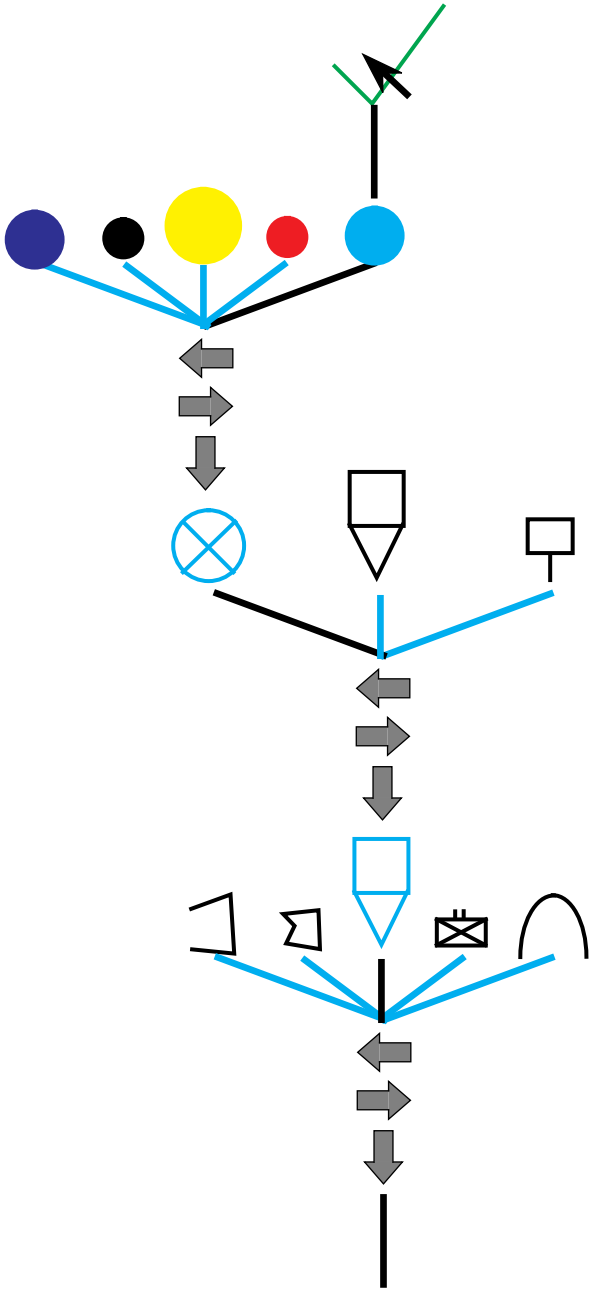


Figure 38: Expansion to Terminator Building a Point

The functions which actually build control measures locally on the Sand Table are contained in the file po_build.cc. Also contained in po_build.cc are a set of state

variables which are global to po_build.cc. These variables are shown in Figure 39.

```
PO_TYPE     buildType;
PO_COLOR    buildColor;
pfVec3      points[C2_SEGMENTS];
            origin;
int         numpoints = 0,
            direction = 0,
            munition,
            detonator;
ushort      dashed = 0,
            style,
            thickness,
            closed,
            splined,
            width,
            beginArrowHead = 0,
            endArrowHead = 0,
            minefieldType,
            size,
            rowWidth,
            density;
```

Figure 39: State Variables in po_build.cc

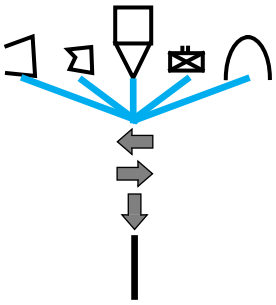
These variables correspond directly to member names within PO_MEASURES_CLASS and the fields of `SP_DescribeObjectVariant`. These variables include the attributes which describe each of the types of control measures. They contain the values of the control measure currently being built by the menu system. In order to understand the variables' use, the Cone Tree menu must be further examined.

Upon selection of an icon from the menu system, the icon and its branch are highlighted and the tree is expanded a level. However, this is only the graphical aspect of an icon selection. In addition to these actions, with each selection of an icon a callback function is invoked. These callback functions are what actually enable the menu system to carry out expected actions for icon selection. During the creation of an object these callback functions are used to set the state variables described above. At each level of the

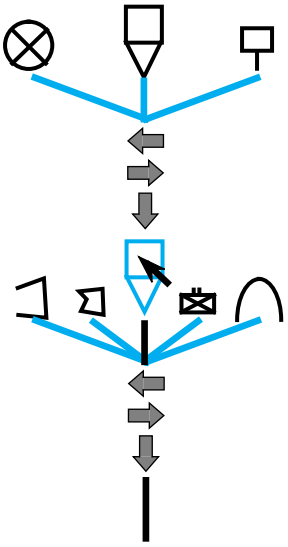
menu some attribute of the object is being set in the state variables via the callback functions.

Using the example of creating a point, when the user selects the general point at the lowest menu level, the Sand Table is informed that the construction of a point has begun. This is done by the callback associated with the general point icon which sets the state variable `buildType` to `PO_POINT` which is one of the types of control measures. When the user selects the coordinating point at the next level, the state variable `style` is set via the icon's callback to `SP_PScoordinating` which defines one of the point styles in ModSAF. Lastly, when a color panel is chosen, the state variable `buildColor` is set in the color panel's callback. Note that only those variables which are applicable to a particular control measure are set. The only needed state variable which was not set directly by user icon selection is the actual location of the point. This state variable is contained in `points`. Since, when creating a point, the stem of the menu is the location at which the point will be placed, `points` is continuously updated whenever the menu is moved. Each time the menu is moved (which may be in the middle of point construction) the function `passMenuPosition` is called. This function updates `points` accordingly. After each of the state variables is set the menu expands to the terminator icon which itself has a callback function named `buildItCB`. The expansion of the tree and associated actions in building the measure are given in Figure 40 and Figure 41.

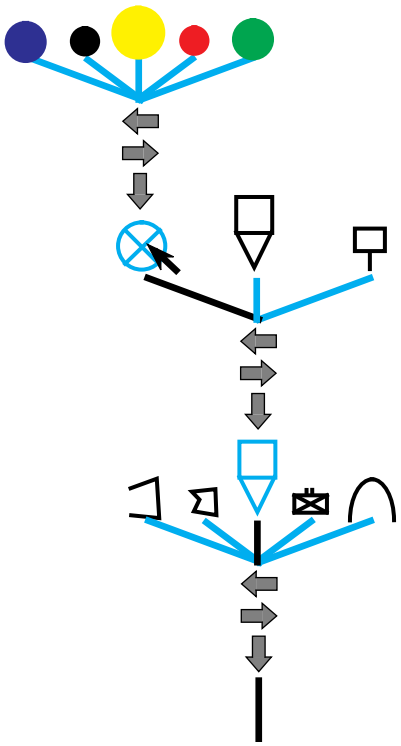
Within the `buildItCB` callback function the control measure is actually built. The function first checks `buildType` which was set during menu expansion and indicates the type of measure to be created. Upon determination of type, the appropriate constructor is called. State variables are sent as constructor parameters and the measure of PO_POINT_CLASS is created. Having created the measure as a PO_POINT_CLASS, the graphical representation of the control measures must be added to the Performer scene tree. Within the PO_POINT_CLASS object is the member `polys` which contains the Performer pfGroup of the object. This pfGroup is added to the Sand Table scene. Next, the newly created object must be put into the Sand Table control measure database. Recalling this



1. Menu Placement. `passMenuPosition` places terrain position in `points` state variable.

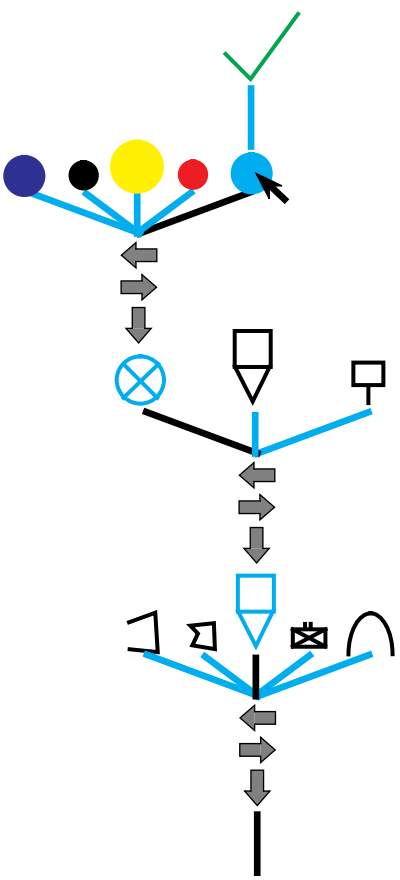


2. Build Point Icon Selected. State variable `buildType` set via icon's callback to `PO_POINT`.

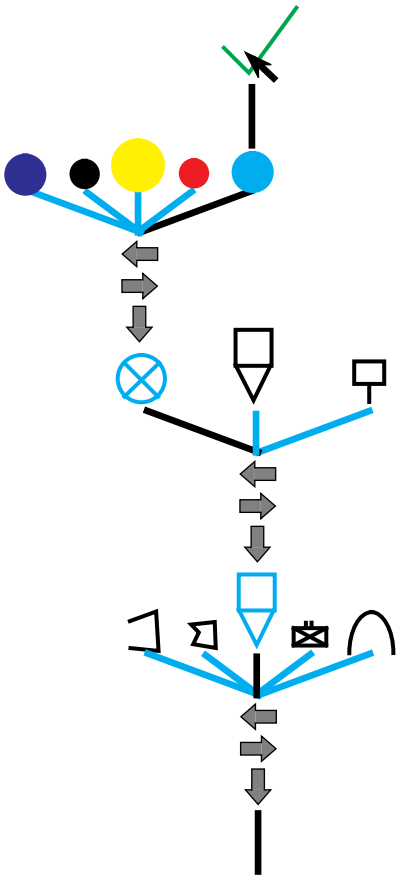


3. Coordinating Point Icon Selected. State variable `style` is set via the icon's callback to `SP_PScoordinating`.

Figure 40: Actions in Building a Measure



4. Color Panel Selected
State variable `buildColor` is set in the color panel's callback.



5. Terminal Icon Selected
State variables have all been set. Call `buildItCB` and build point.

Figure 41: Actions in Building a Measure (Continued)

from the above sections (see page 56), to be included in the database an object needs a pointer to a PO_MEASURES_CLASS which is created using the constructor and a key which consists of the `objectID`.

The `objectID` is constructed using the site and host of the Sand Table simulator. The object number portion of `objectID` is selected from the variable `object` which is contained in po_funcs.cc. With this data `objectID` is created and the measure is inserted into the database of control measures. The final action needed in the creation of

the object is sending the new object description to the network. This is accomplished by `updatePDU` and `sendPDU` which were both described above (see page 67).

b. Line Creation

The creation of linear control measures differs from that of point creation only in the graphical building of the line and the actual constructor used to create the measure. The obvious difference arises in that a line consists of many points rather than just one. The construction of a line can best be understood by again following an example of building a line using the menu. From the base level of the Cone Tree menu, the open line icon is chosen. The menu expands one level, the open line icon and branch are highlighted and the callback function sets `buildType` to PO_LINE. Additionally, a flag is set which enables the selection of locations on the terrain. This flag is in po_build.cc and is named `addingLine`. This expansion can be seen in Figure 42.

At the next level in the Cone Tree menu are a selection of line segments each of different thickness. Selection of a thickness determines how thick the line being constructed will be. Selecting a line thickness expands the menu and sets `thickness` via the callback function. The expansion is shown in Figure 43.

The next level of expansion is again a selection of circles each containing a different color. The selection is the same as with point construction expanding the menu to the terminator and setting `buildColor`. This is shown is Figure 44.

The construction of a line diverges from that of a point at this time. When `addingLine` is set, the left mouse button is enabled to pick terrain. When the user clicks on terrain a small cone is placed on the terrain at the location selected to indicate the point. When more than one point is selected a line is drawn over the terrain connecting the points. This is shown in Figure 45.

As each additional point is added the line is extended to include the point. Each of the cone point indicators can be dragged and dropped to change the position of the line. Further, the entire line can be dragged and dropped to a new location. These graphical

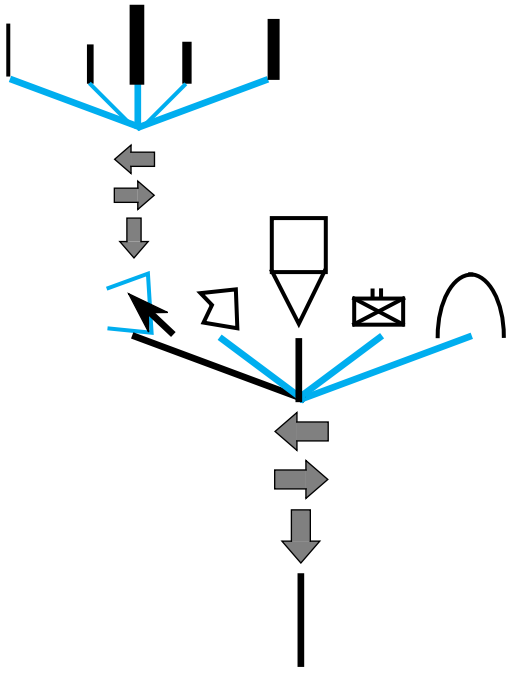


Figure 42: First Expansion Building a Line

concepts will be covered in Chapter V. The important characteristic to be considered here is the actual building of the line. As each point is placed on the terrain, the location is put into the state variable `points` and the state variable `numpoints` is incremented. When the terminator is selected `buildItCB` is again called. Based on `buildType` the constructor for a PO_LINE_CLASS is invoked. The measures graphical representation and PO_MEASURES_CLASS representation are added into the Sand Table in the exact same way the point measure was added. Similarly, `updatePDU` and `sendPDU` are called to send the new control measure to the network.

The creation of a closed line and minefield are the same as the construction of a line with the exception of the last point being connected to the first point of the line. Lastly, the creation of text from the Sand Table is not available in the current implementation.

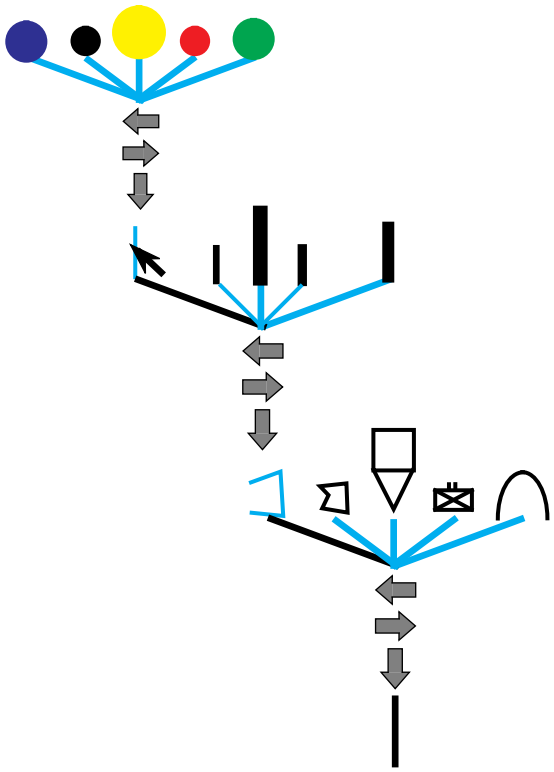


Figure 43: Second Expansion Building a Line

7. Proper Display of Control Measures on the Sand Table

The actual display of the control measures is accomplished by creating a Performer pfGroup for each control measure known by the Sand Table. Within each PO_MEASURES_CLASS object is the member `polys` which contains a pointer to the pfGroup representing the control measure. These pfGroups are then added to `L_po_meas` which is contained in po_funcs.cc. `L_po_meas` is a pfGroup itself and is the parent of all control measures in the scene graph.

The display of the control measures can be toggled by using the function `toggle_po_display`. When the function is called, it determines whether or not the

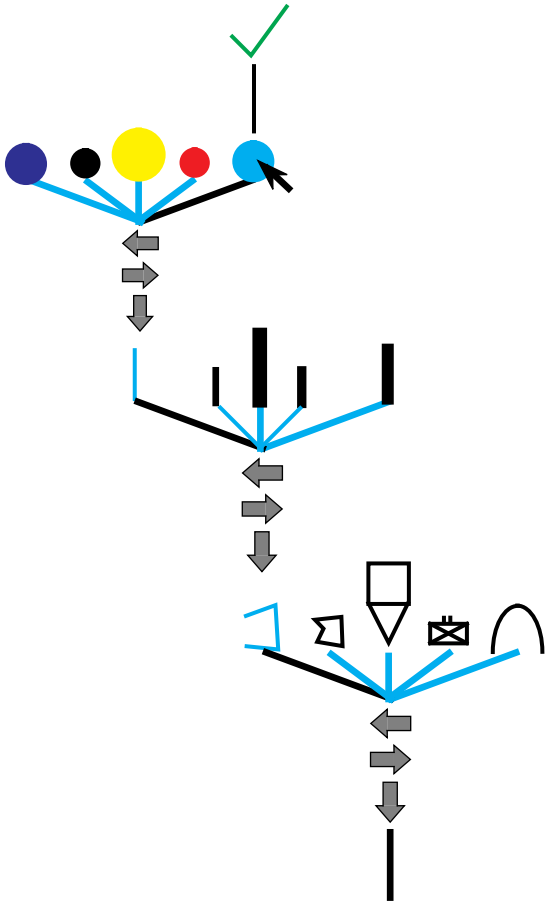


Figure 44: Expansion to Terminator Building a Line

control measures are currently being displayed. If so, the function `po_display_measures` is called and if not the function `po_hide_measures` is called. These functions are straightforward with `po_display_measures` adding `L_po_meas` to the NPSNET scene graph and `po_hide_measures` removing (but not deleting) `L_po_meas` from the scene graph.

The adding of `L_po_meas` to the NPSNET scene graph and the adding of the pfGroups represented in the member `polys` is the only graphical connection between the Sand Table and NPSNET IV.8. The additional pfGroups and pfGeodes which represent the control

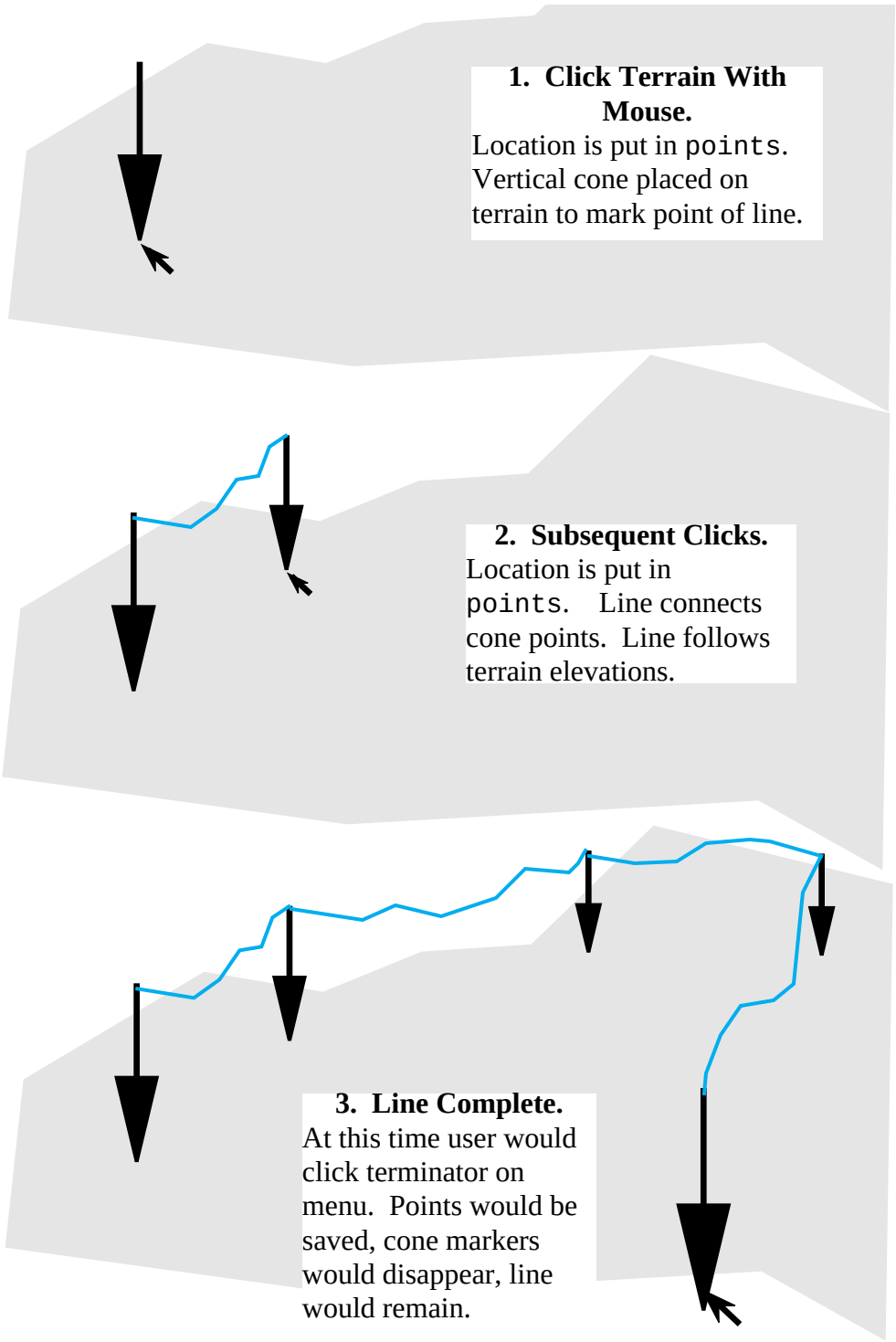


Figure 45: Point Indicator During Line Construction

measures are part of the normal Performer graph traversal. This is not meant to say that the graphical portion of the Sand Table is trivial; rather, it should be noted that `L_po_meas` is the only "hook" into the NPSNET scene graph. Further, no changes in the control measures will ever have an effect above `L_po_meas` in the scene. The actual graphical implementation of the Sand Table *below* `L_po_meas` is very extensive and will be covered in detail in Chapter V.

8. Manipulation and Movement of Control Measures on the Sand Table

The final task which the Sand Table must accomplish is the moving and manipulation of control measures. Control measures can be moved on other simulators and their movement is reflected on the Sand Table. Similarly, objects can be dragged and dropped on the Sand Table and PDUs will be broadcast to reflect the movement. The movement of an object by another station will be seen at the local Sand Table only through the receipt of `SP_DescribeObjectVariant` PDUs. The receipt of these has already been considered but basically the member values of the object will be updated to reflect any change which the PDUs may indicate. The member function `update` will use the incoming PDU to change any members of the object needing update. Any geometry changes will also be accomplished with modification of the `polys` pfGroup.

The manipulation and movement of control measures by the local Sand Table is a much more intensive task. Currently only movement of the control measures is implemented in the Sand Table. In this case the factors to be considered were the "picking" of control measures, the dragging of control measures, the dropping of control measures, the updating of the PO_MEASURES_CLASS object, the updating of `SP_DescribeObjectVariant` PDUs and the sending of `SP_DescribeObjectVariant` PDUs.

The updating and sending of the PDUs is accomplished in the same way the PDU was created when a new object was created on the Sand Table. This is accomplished by `updatePDU` and `sendPDU` which were both described earlier (see page 67). The

updating of the PO_MEASURES_CLASS object is accomplished as part of the dropping of the control measure. When dropped the member `points` (see page 50) is modified to reflect the dragging of the measure. The member `polys` is also changed to reflect the new terrain over which the measure is located. Currently, the measures can only be dragged so only `points` and `polys` change. The picking, dragging and dropping of measures will be covered in more detail in Chapter V.

C. SUMMARY

This chapter gave an operational and design overview which highlighted basic elements of functionality which were designed and constructed. The Sand Table representation of control measures was discussed with the examination of the PO_MEASURES_CLASS C++ class which is used to maintain objects. The maintaining of the database of control measures utilizing an array was examined. The receipt and processing of PDUs using functions in po_funcs.cc was examined to include capturing of Simulator Present PDUs, Delete Object PDUs and Describe Object PDUs.

The sending of PDUs utilizing the polymorphic member functions `updatePDU` and `sendPDU` was examined. Also, the sending of "handshaking" PDUs was discussed. The local creation of control measures was examined using examples of object creation using a cone tree menu system. Lastly, the display, manipulation and movement of control measures was introduced and will be covered in more depth in Chapter V.

V. CONTROL MEASURE VISUALIZATION AND MANIPULATION

A. INTRODUCTION

Having discussed the major design and implementation considerations, the specific graphical considerations will now be examined. A goal of the Sand Table was the intuitive placement and manipulation of control measures which will depict the plan being constructed (see page 7). While much of the control of PDUs and maintenance of control measures has been covered, the actual 3D graphical interface aspects of the Sand Table must now be examined.

In the current version of the Sand Table points, lines and minefields can be visualized and manipulated. Each will be examined and detail; however, some higher level design characteristics must first be considered. Placement of measures was chosen to correspond to a logical and visually appealing method. Measures placed on the terrain are easily seen yet do not overwhelm the scene. This is accomplished by making the measures the correct size and by appropriately orienting the measures with regards to the terrain.

Point control measures are placed "on the ground." This is the most logical choice since a point control measure will refer to a specific point on the terrain. The points are 2D objects which are "billboarded" to face the viewer. Care must be taken so that the points will not sink into the ground nor hover over the ground. The "sinking" would be visually unappealing and the "hovering" would create an ambiguous cue as to where the measure was actually located over the ground. While the objects are represented with a 2D icon the visualization is sufficient and appealing. As was stated earlier (see page 17) the symbols used are already defined and have been used on overlays and other two dimensional methods of portraying control measures. It is questionable whether a 3D representation of these familiar icons would better the visualization. Where the Sand Table enhances the display is that the familiar 2D icons are *presented* and *manipulated* in the 3D virtual

environment. The “billboarding” ensures that the familiar icon view is always oriented correctly towards the viewer. Further, as the distance between the viewer and icon changes, the size of the icon behaves as expected in that it becomes smaller when viewed from a greater distance. The result of this design is that there is no ambiguity as to what military symbol the icon represents nor is there ambiguity as to the exact position of the point on the terrain. This is shown in Figure 46. How this visualization is actually accomplished will be covered below in Section B.

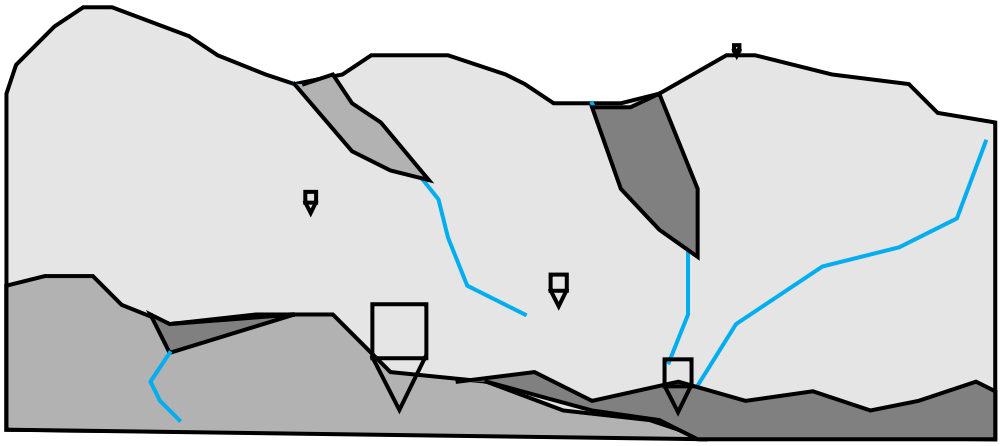


Figure 46: Point Placement on Terrain

The two linear control measures, lines and minefields, are represented in a different manner. Unlike points, lines and minefields currently “hover” over the terrain. While this created an ambiguous location in the case of a point this is not the case for linear measures. This is due to the nature of a linear control measure which allows the eye to “fix” a line over terrain. The line is taken as a whole and the mind correctly correlates the linear

measure's position. This is also enhanced with the way lines are placed over the terrain. In the Sand Table linear control measures are "terrain following."

Contrasting a point which is placed at a single point hence, a single elevation, a linear control measure consists of many points and varying terrain altitudes. The linear control measures are placed to follow these elevations with the height at which the measure "hovers" being constant. The linear measures follow the contours and folds of the terrain. This is shown in Figure 47. Using a physical analogy, the linear measures are placed like an electrical power line over hilly terrain. The power line maintains a constant height above the ground. The contouring allows for unambiguous placement of the lines with the terrain itself giving cues as to where the linear measure is located. The placement also allows for better visibility of the line from a variety of altitudes and viewer orientations. Lastly, the placement eliminates the need to deal with Z-buffer problems which would occur if the line and terrain were placed at the same position.

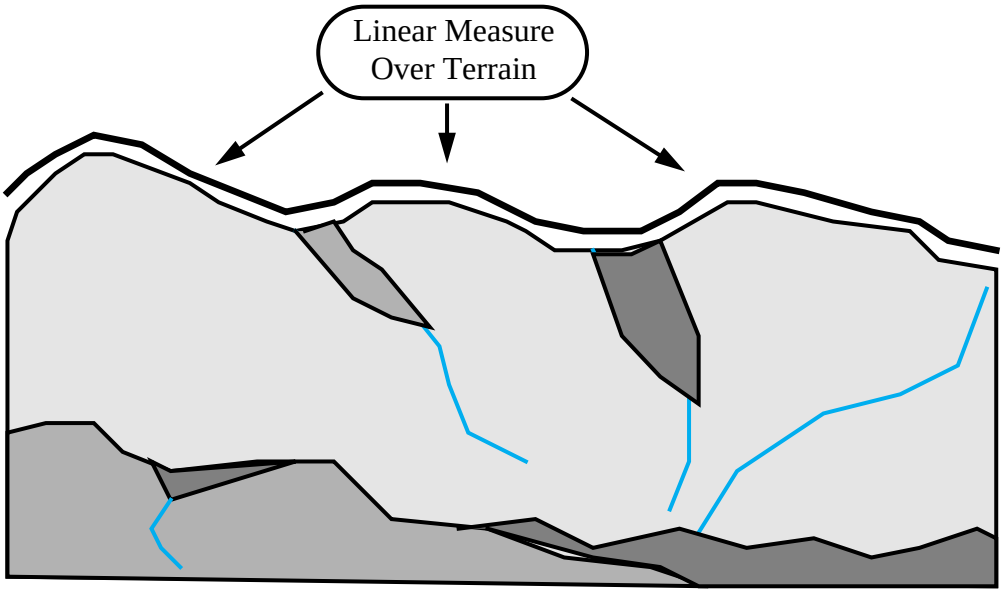


Figure 47: Line Placement Over Terrain

Once visualized, the Sand Table must also have the means to manipulate the control measures. Various options were considered. One option was using a 3D mouse cursor to select positions on the terrain and select control measures themselves. The cursor would "float" through the air above the terrain appearing as a 3D object with size and orientation being based on viewer perspective. This was deemed unnecessarily complicated in the initial application of the Sand Table since all of the mouse functionality would have the natural constraint of either referencing terrain or actual objects. However, this should not suggest that such a system will not be of utility in future development. Of particular note would be a stereoscopic application, in which a true 3D mouse cursor would have to be revisited.

The method actually implemented was a 2 1/2D mouse cursor system. The term "2 1/2D" refers to the fact that the mouse only moves in two dimensions; however, its placement on terrain or on an object gives the third dimension *implicitly*. This is shown in Figure 48.

The use of the 2 1/2D cursor offers several advantages. The first is its simplicity. The user need not learn any special mouse functionality. The mouse and its cursor, functions as a user familiar with a PC or workstation mouse expects. Next, by using the mouse in this manner, programming of mouse input is simplified by using Iris GL's existing mouse functionality which includes determining mouse screen position and mouse button status. The last benefit of using the 2 1/2D mouse is the ability to use Performer picking functionality. "Picking" refers to the selecting of a piece of the Performer scene using mouse position. Especially useful is the ability in Performer to query a pick to return the pfGeode or pfGroup which was selected by a mouse location. These specific Performer aspects will be considered in detail below. As will be shown, while offering a great deal of capability, the Performer "picking" functionality was not without unexpected implementation difficulties.

Using the 2 1/2D mouse system the capability of a "hot mouse" was included in the design of the Sand Table interface. The term "hot mouse" refers to some indication being

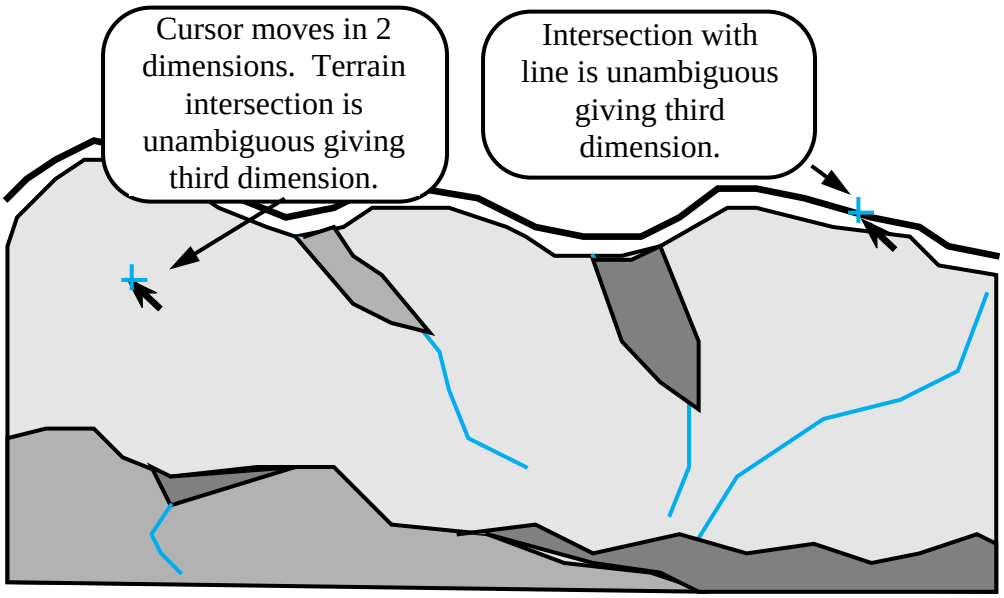


Figure 48: 2 1/2D Cursor Implementation

given to the user when the mouse cursor is over an object, in the Sand Table which may be manipulated, without a mouse button having to be depressed. Specifically, when the cursor passes over a control measure on the Sand Table the measure is highlighted with a different color, currently a bright pink color. When the cursor is no longer located over an object the object's color returns to its original color. When the cursor is located over terrain or sky nothing is highlighted. The "hot mouse" gives visual feedback to the user as to which control measure is currently being selected with the user then being able to select a mouse button which will manipulate the measure. The "hot mouse" is shown in Figure 49, Figure 51 and Figure 50.

B. GRAPHICAL MEASURE IMPLEMENTATION

Having presented an overview of the actual interface functionality of the Sand Table, the specific implementation will be examined. Recall, each control measure is represented

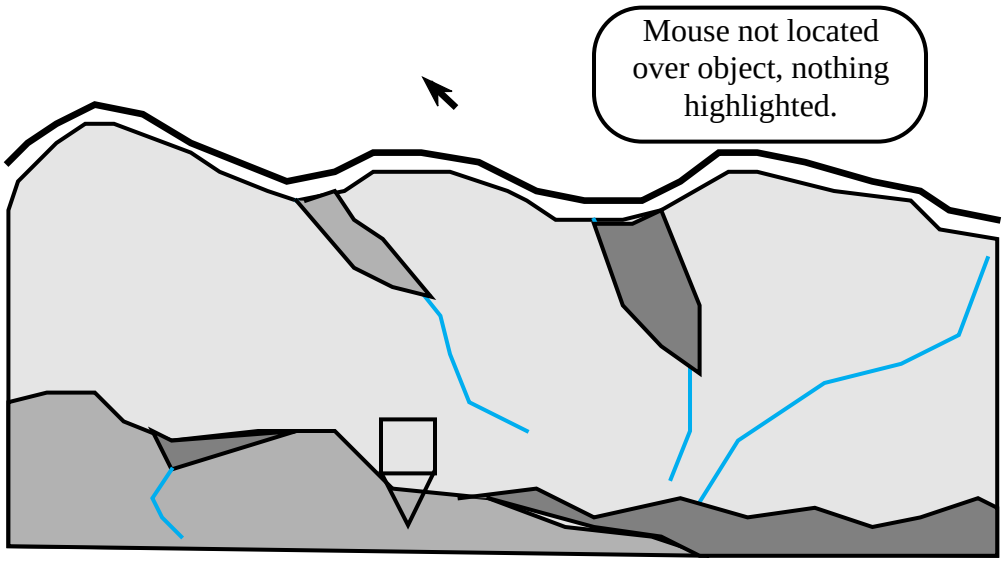


Figure 49: Hot Mouse Object Highlighting

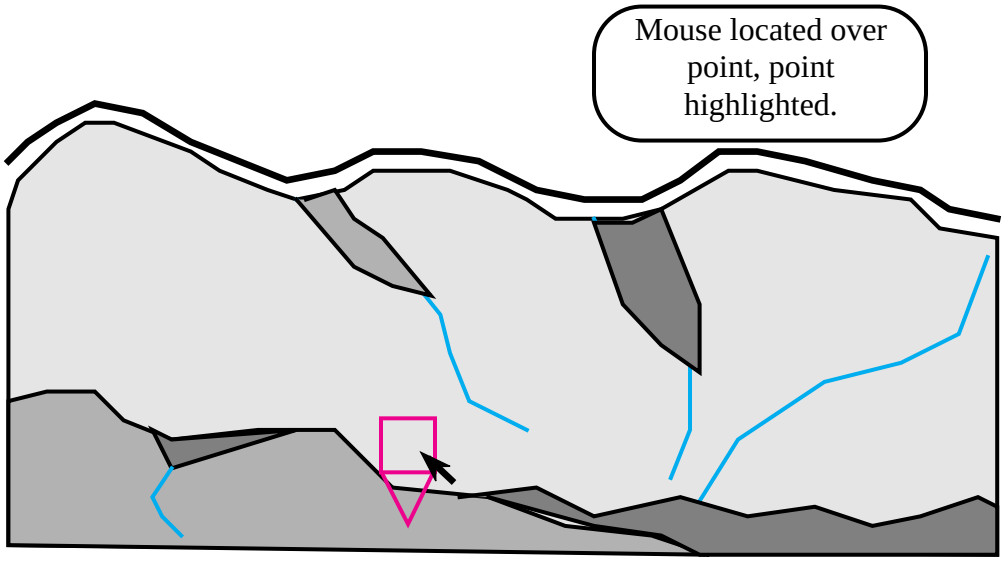


Figure 50: Hot Mouse Point Highlighting

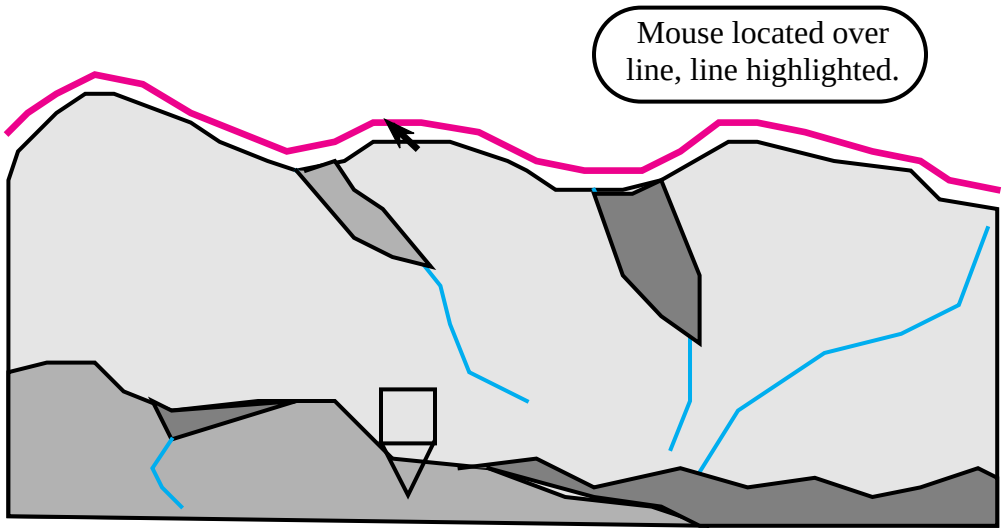


Figure 51: Hot Mouse Line Highlighting

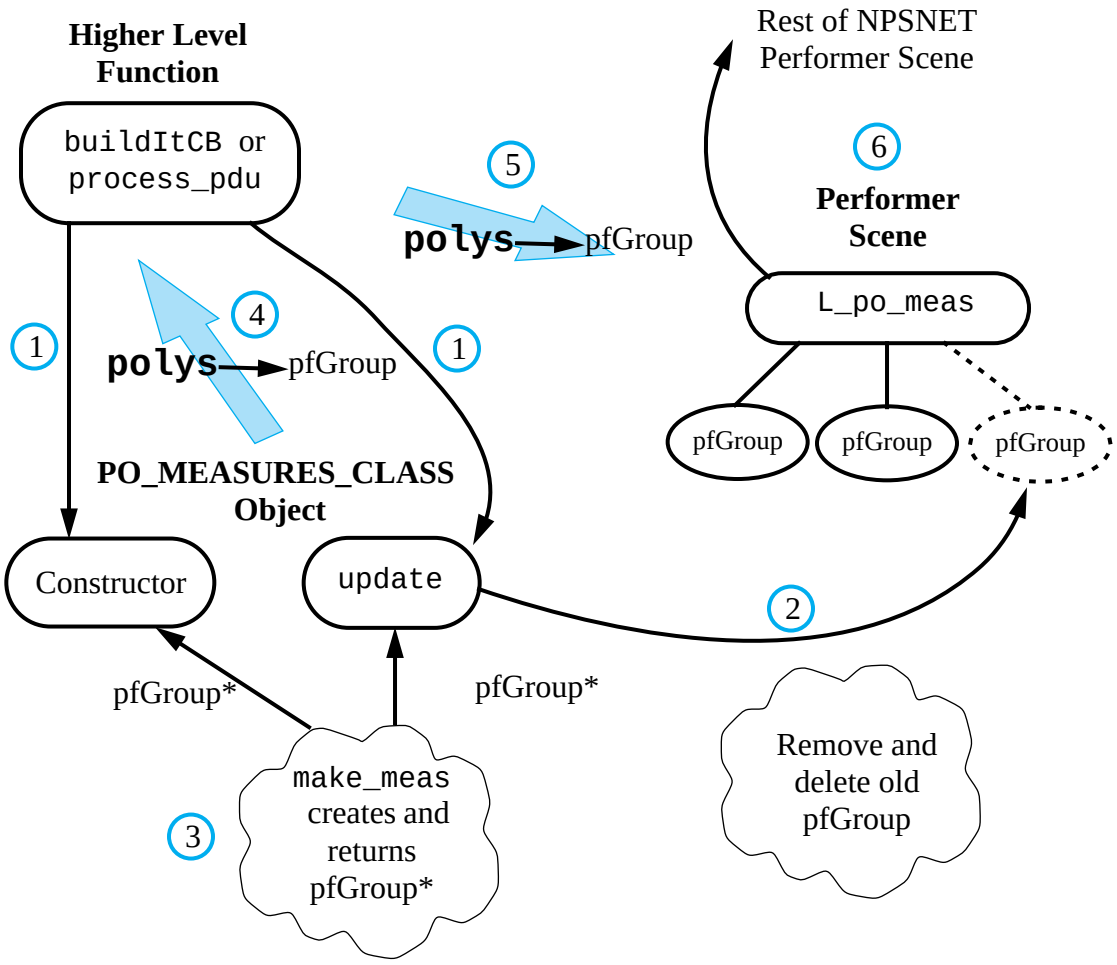
in three different ways (see page 49). The first is how it is stored within the PO_MEASURES_CLASS, the second is how it is stored in the `SP_DescribeObjectVariant` PDU and the last is how the measure is actually displayed with Performer. The graphical interface of the Sand Table is concerned with the last of these and uses the first two to actually construct the last.

As discussed in Chapter IV, the PO_MEASURES_CLASS (see Figure 21) a member `polys` of type `pfGroup*` is included in the class. The pfGroup which `polys` points to is the graphical representation of that measure. So in essence each object stored in the Sand Table database has a pointer to the pfGroup which represents itself. Within the constructors of each of the derived classes of PO_MEASURES_CLASS, the members defining the state of the control measure are assigned either by extracting the appropriate fields from the `SP_DescribeObjectVariant` PDU or the values are passed directly via parameter.

Additionally, within the constructor is the assignment of a pointer to the pfGroup, to the member `polys`. This is accomplished by assigning the return value of the member function `make_meas` to `polys`. `polys` is then available for inclusion on the Performer scene graph which is done by adding the pfGroup to `L_po_meas`. See “Proper Display of Control Measures on the Sand Table” on page 82.

An overall flow in the creation of a control measure from a graphical standpoint can now be considered. First, a higher level function such as `buildItCB` (see page 77) or `process_pdu` (see page 58) will need to create an object of type PO_MEASURES_CLASS. When the constructor is invoked `make_meas` will create the pfGroup which represents the object and a pointer to the group will be stored in `polys`. The higher level function will then have access to the pfGroup through `polys` to add the pfGroup to `L_po_meas`. Another possibility is the receipt of a `SP_DescribeObjectVariant` PDU which is a modification of an object already in the Sand Table database. In this case the member function `update` (see page 51) is called which removes the old pfGroup from `polys` and assigns a new pfGroup via `make_meas` to be used by the higher level function. Similarly, if an object were deleted, its pfGroup would be removed and deleted via the `polys` pointer. The deletion would be accomplished by `delete_object` (see page 58). This flow in which the graphical portion of an object is managed is depicted in Figure 52.

`make_meas` is where most of the desired functionality given in the graphical overview is accomplished. `make_meas` actually constructs the Performer geometry which is rendered in the Performer draw process. Since each type of control measure is constructed differently, the placing of `make_meas` as a member function within the PO_MEASURES_CLASS derived classes is another logical use of polymorphism. The actual implementation of `make_meas` for each of the derived classes is covered in detail in the next sections. Descriptions of the Performer functions and types used can be found in the IRIS Performer Programming Guide and in subsequent C++ excerpts performer functions begin with the prefix “pf”.



Graphical Flow. 1. Higher level function calls constructor or `update`. **2.** If `update` called remove and delete old pfGroup. **3.** Build and return new group using `make_meas`. **4.** pfGroup now accessible through member **`polys`** pointer. **5.** Via **`polys`** pointer, attach pfGroup to `L_po_meas` as child. **6.** `L_po_meas` is attached to NPSNET scene graph.

Figure 52: Graphical Creation Flow of Control Measures

C. POINTS

The current Sand Table implementation has the ability to represent three different kinds of points from the possible seventeen styles of points (see page 34) in ModSAF. The

points include the general checkpoint, the coordinating point and the contact point. These three points are shown in Figure 53. Only three points are currently represented; however, the underlying architecture can support any number of points and the current Sand Table versions serves as a proof of concept. The actual construction of the points is accomplished in the PO_POINT_CLASS `make_meas` which is contained in po_funcs.cc.

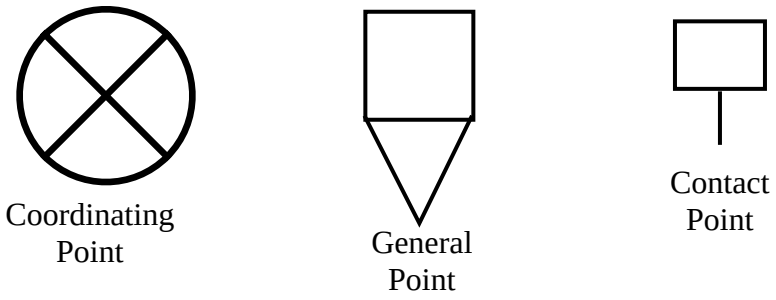


Figure 53: Types of Point Measures Represented

The actual geometry of the points is contained within a pfGeoSet which is added into a pfBillboard to achieve the desired "billboarding" effect. The billboard effect ensures that the point measure always faces the viewer. This structure, along with using pfBboardPos to position the points, initially seemed sufficient to represent a point measure. Unfortunately, due to desired functionality and Performer 1.2 documented deficiencies, the final structure used is much more complicated. If subsequent versions of the Sand Table are written using Performer 2.0, the structure may be more simplified and closer to that described above.

The deficiencies are in two areas. The first is the inability to use the Performer picking functions on points, lines or other pfGeoSets which are in billboards. Since a goal in the graphical functionality was to "pick" objects with a 2 1/2D mouse cursor, this deficiency

presented a problem. The point measures were constructed using line strips and were placed in billboard structures, resulting in the loss of the ability to pick the measure.

The second deficiency is Performer's inability to "highlight" geometry within a pfBillboard. Performer has a type, pfHighlight, which enables a node to be "highlighted", i.e. rendered in a color different than specified in its pfGeoState, simply by referencing the appropriate pfNode for highlighting and marked the node during scene traversal. This deficiency created a problem with the desired "hot mouse" graphical functionality since the point measures were to be placed inside of billboard structures. The resulting structure is less elegant than desired; however, it successfully works around the Performer deficiencies and implements the desired functionality. The structure is shown in Figure 54.

The structure shown in Figure 54 solves both the "highlighting" and the "picking" problems. In order to solve the "picking" problem, an additional structure had to be added to the point which was not a child of a pfBillboard. To accomplish this the pfDCS was added to the structure of the point measure. The Performer "picking" functions *can* pick through a pfDCS. In order to "pick" a point measure the transparent backboard is actually the pfNode which is being "picked". The Performer graph is then followed backwards using pfGetParent to find the parent group which represents the point. Picking will be discussed later but of importance here is the ability of Performer to "pick" the transparent polygon which is collocated with the visible geometry of the point measure.

In order to solve the "highlighting" problem the use of the organic pfHighlight had to be abandoned. In order to achieve point measures which change to a highlighted color the pfSwitch in the figure was used. A pfSwitch allows the option of rendering selected children of the switch, all of the children of the switch or none of the children. In the case of the point, the children represent the point geometry rendered in the actual color of the measure, i.e. the desired color of the measure on the Sand Table, or the highlighted color. Only one of the children is rendered at any time. The effect is that the measure appears to change color when highlighted when in fact a different pfGeoSet is actually being rendered.

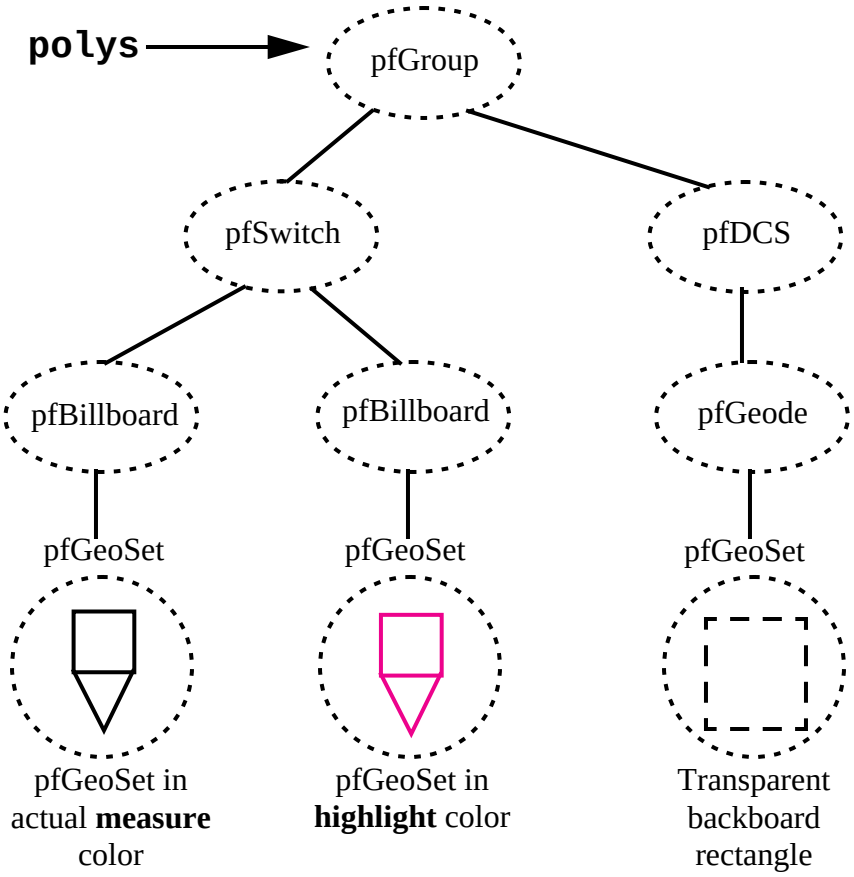


Figure 54: Point Measure Performer Structure

Note, that the point measures' pfGeoSets are still children of pfBillboards so the "billboard" effect is retained.

Next examined is the actual building of the pfGeoSet which contains the geometry of the point. Point measures are constructed using line strips to actually represent the point icon. Performer has the ability to construct line strip primitives. To accomplish this, the

attributes of the pfGeoSet are set using pfGSetAttr. A section of the Sand Table's C++ code illustrates how the pfGeoSet is being constructed and is presented in Figure 55.

```
gset = pfNewGset(pfGetSharedArena());
pfGSetAttr(gset,PFGS_COORD3,PFGS_PER_VERTEX,pointVerts, NULL);
pfGSetAttr(gset,PFGS_COLOR4,PFGS_OVERALL, color, NULL);

pfGSetPrimType(gset, PFGS_LINESTRIPS);
pfGSetNumPrims(gset, numPrims);
pfGSetPrimLengths(gset, length_vec);
```

Figure 55: Creation of Point Measure pfGeoSet

`gset` is of type pfGeoSet in the figure above. The important attributes which are set are `PFGS_COORD3` and `PFGS_COLOR4`. The `pfGSetAttr` function call which contains `PFGS_COORD3` sets the actual vertices of the line strip being drawn. An array of pfVec3, `pointVerts`, is supplied as a parameter in this pfGSetAttr function call. The `pfGSetAttr` function call which contains `PFGS_COLOR4` sets the actual color of the pfGeoset with `color` being a pfVec4 which contains the RGBA vector of the desired color. `pfGSetPrimType` sets the actual Performer primitive type to be used in construction of the pfGeoSet, in this case `PFGS_LINESTRIPS`. When constructing the transparent backboard the primitive type is set to `PFGS_QUADS` which builds a quadrilateral polygon rather than a line strip.

`pfGSetNumPrims` sets the number of primitives which will be included in the pfGeoSet. More specifically, when using the above method to construct a line strip, a pfGeoSet may contain several *unconnected* line strip *segments* to represent the desired geometry. An analogy would be drawing a picture with a pencil on a piece of paper. Each time the pencil is lifted from the paper a line strip *segment* is left on the paper which comprises one part or primitive of the entire drawing. Referring to the coordinating point

in Figure 53, the corresponding pfGeoSet for the coordinating point would consist of three primitives, one segment for the circle, and one for each of the lines comprising the “X”.

Lastly, `pfGSetPrimLengths` sets the total number of vertices used in the pfGeoSet’s line strips. With the actual construction of the pfGeoSet being outlined, the remaining consideration is how `color`, `pointVerts`, `numPrims` and `length_vec` are given the correct values for the point being constructed. Recalling the PO_MEASURES_CLASS, `color` is already a member of the class (see Figure 21) and its value should be set prior to calling `make_meas`. The next three parameters used in the construction of a point are contained in a file which stores the geometry for each point. Since the geometry of point measures is relatively simple, point geometry is created in advance for each style of point and stored in a file for future use. The file format is simple containing the number of primitives, the number of vertices in each primitive and a list of the x, y and z coordinates for each vertex. The file is stored as a text file with the suffix “.lst” (line strip).

The total functioning of `make_meas` can now be examined. When called, `make_meas` examines `style` of the PO_POINT_CLASS object. Based on this style the appropriate file is opened and `numPrims`, `length_vec` and the actual vertices are read. Three pfGeoSets are then created, the first is created using the code of Figure 55. The next uses the same code except for the setting of `PFGS_COLOR4` in which a pfVec4 for the desired highlight color is supplied as a parameter. Lastly, a pfGeoSet is created using a primitive type of `PFGS_QUADS` rather than `PFGS_LINESTRIPS` and a “.lst” file containing vertices for the transparent polygon. A transparent color is also supplied.

The first two pfGeoSets are then added to the pfBillboards and the last pfGeoSet is added to the pfGeode which is added to the pfDCS of Figure 54. The rest of the structure of Figure 54 is then constructed by simply creating the shown pfNodes and adding the corresponding children. The final consideration is the actual location of the measure. Recalling PO_MEASURES_CLASS (see figure 21 on page 50) the location of the point measure is contained in the member `points`. This location is used to position the two

pfBillboards and the pfDCS which are updated to be coincident with `points` ensuring the point geometry and the backboard used for picking are collocated.

This completes the construction of the point measure which is contained in the pfGroup shown in the Figure 54. The pointer to this pfGroup is then returned and assigned into `polys`.

D. LINES

The next version of `make_meas` to be examined is that used to create the Performer representation for a linear control measure. The current version of the Sand Table only implements one style of line; however, the thickness and color can be varied. Ideally, the representation of the line would consist of a single pfGeode which contained a pfGeoSet for a single line strip over the terrain. However, the Performer limitations which created problems in "highlighting" and "picking" of points have the same effects on lines although to a lesser extent. Linear control measures do not use pfBillboards but they are constructed using line strips which precludes immediate "picking" and "highlighting" using Performer functions. Another consideration is that unlike points, which have the same geometry and are simply moved using billboard position and DCSs, every line has a unique geometry describing it and in order to accomplish "terrain following" graphical functionality, as described above, the geometry of a line will change each time it moves. The result of this consideration is that every duplicative pfGeoSet used to correct for "picking" and "highlighting" may have adverse consequences on performance due to the recurring construction of lines.

The resulting Performer structure is shown in Figure 56. While not as complicated as the Performer structure for a point, the actual construction still has a redundant representation of the measure which has a slight performance penalty.

Comparison of the linear structure with the point structure can assist in understanding the structure. The first notable differences are the lack of pfSwitchs, pfBillboards and pfDCSs. Switches are not used because while a duplicate line could be made in the

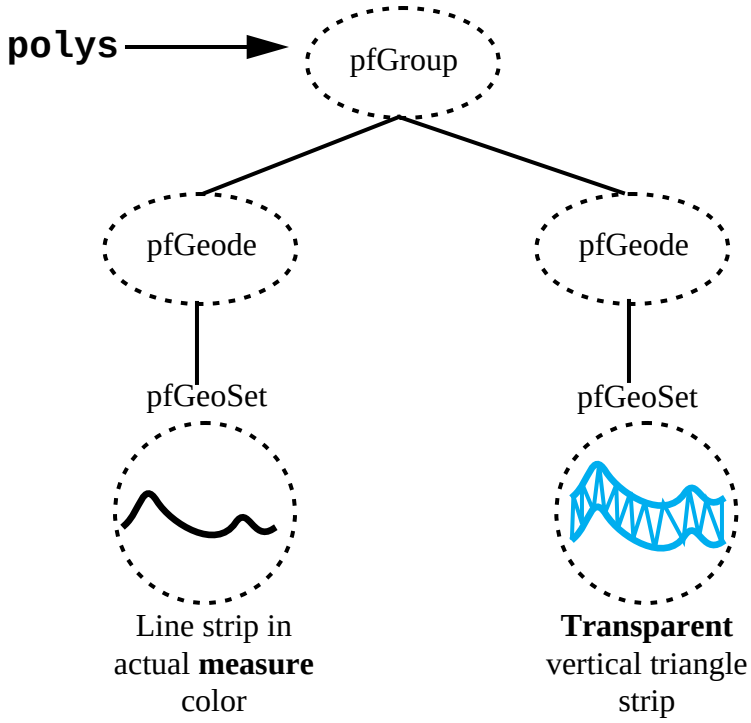


Figure 56: Linear Measure Performer Structure

highlight color, the performance penalty would be too great. This is due to the fact that even though the additional line strip would solve the "highlighting" problem a line strip still cannot be "picked" so an *additional* structure would still have to be added resulting in three representations of the same line. The above structure results in only one duplicate pfGeoSet being created rather than two.

pfBillboards are not needed because lines are not rotated with respect to the viewer and are placed *absolutely* over the terrain. Lastly, pfDCSs are not needed because the actual point locations on the terrain are used in the construction of the line strips in the pfGeoSet. This differs from the line strips used for the points in which the icon was drawn using a *relative* coordinate system and then translated to a location on the terrain.

The linear measure structure presented in Figure 56 solves the “picking” and “highlighting” problem by using the fact that while a line strip cannot be “picked” a triangle strip can be “picked”. Hence, while the pfGeode containing the line strips is used to actually render the line, the transparent triangle strip is used to query a mouse hit. The query will return the pfGeode containing the triangle strip and the Performer scene graph can be followed backwards to the parent pfGroup as was done with the point Performer structure.

In order to highlight the “picked” group the actual color attributes of the pfGeoSet are changed dynamically. This is done in the function `poHighLight` which is contained in the file highlight.cc. This is accomplished by using `pfGSetAttr` each time the line is highlighted. `pfGSetAttr` is invoked setting `PFGS_COLOR4` to the desired highlight color. The old `PFGS_COLOR4` is retained and `pfGSetAttr` is again invoked when the measure needs to return to its actual color. This solution may also seem applicable to the point measure eliminating the need for a duplicate pfGeoSet; however, it was discovered while programming the measures, that the changing of color’s attributes did not work when the pfGeoSet was the child of a pfBillBoard.

Having described the structure in which the linear control measure is represented, the actual construction will be examined. An excerpt from the PO_LINE_CLASS member function `make_meas` will again serve as a good starting point and is presented in Figure 57. The code is nearly identical to that presented for the point measure with some exceptions which will be discussed.

`gset` is again of type pfGeoSet and stores the actual geometry of the line. `coords` is a pointer to an array of pfVec3 which contain the x, y and z coordinates of the vertices in the line. `color` is a pfVec4 member value of the class and should have been set prior to calling `make_meas`. The primitive type is still `PFGS_LINESTRIPS`; however, the number of primitives is only one since the pfGeoSet will consist of only one connected line strip. `length_vec` contains the number of vertices in the line strip. Lastly, `pfGSetLineWidth`,

```
gset = pfNewGset(pfGetSharedArena());
pfGSetAttr(gset,PFGS_COORD3,PFGS_PER_VERTEX,coords, NULL);
pfGSetAttr(gset,PFGS_COLOR4,PFGS_OVERALL, color, NULL);

pfGSetPrimType(gset, PFGS_LINESTRIPS);
pfGSetNumPrims(gset, 1);
pfGSetPrimLengths(gset, length_vec);
pfGSetLineWidth(gset, thickness);
```

Figure 57: Creation of Linear Measure pfGeoSet

as the name implies, sets the width of the actual line strip in pixels using the class member `thickness` which like `color` should be set prior to calling `make_meas`.

What remains to be determined is `coords` and `length_vec` which are interrelated and determined as follows. Unlike point measures, whose geometry is known a priori, the number of vertices and actual vertex values must be calculated for each line. Recalling the PO_MEASURES_CLASS (see Figure 21), the member `points` contains the vertices of the line. However, these vertices are not enough to construct a line because their granularity is not small enough. More specifically, when constructing a line, the user places points at where the line changes direction (see Figure 45). This is not based on terrain elevation; rather, it is only based on geographic position.

In order to achieve a *perfect* "terrain following" functionality for the lines, the actual elevation of the terrain must be known at *all* points on the terrain under the line. These elevations would then be used to elevate every point of the line strip the proper distance above the terrain. While a *perfect* line strip is impractical, the concept shows that more elevation samples are needed under the line than those of the locations chosen by the user and stored in `points`.

An example will clarify the concept and is presented in Figure 58 and Figure 59. Figure 58 illustrates a line constructed using only the vertices given in `points`. As can be

seen, the given points are not sufficient to make a smooth line over the terrain. The line actually goes through terrain because the elevations simply connect the vertices of `points`. Figure 59 takes each line segment between two vertices in `points` and subdivides the segment into smaller segments. The result is a sequence of elevation "posts" and a closer "terrain following" representation. Note that in flat terrain fewer posts are required than in steep mountainous terrain. Additionally, a lower resolution line (fewer elevation "posts") can be used when moving or manipulating a line, which is exactly what is done when the Sand Table drags a line. This will be examined later.

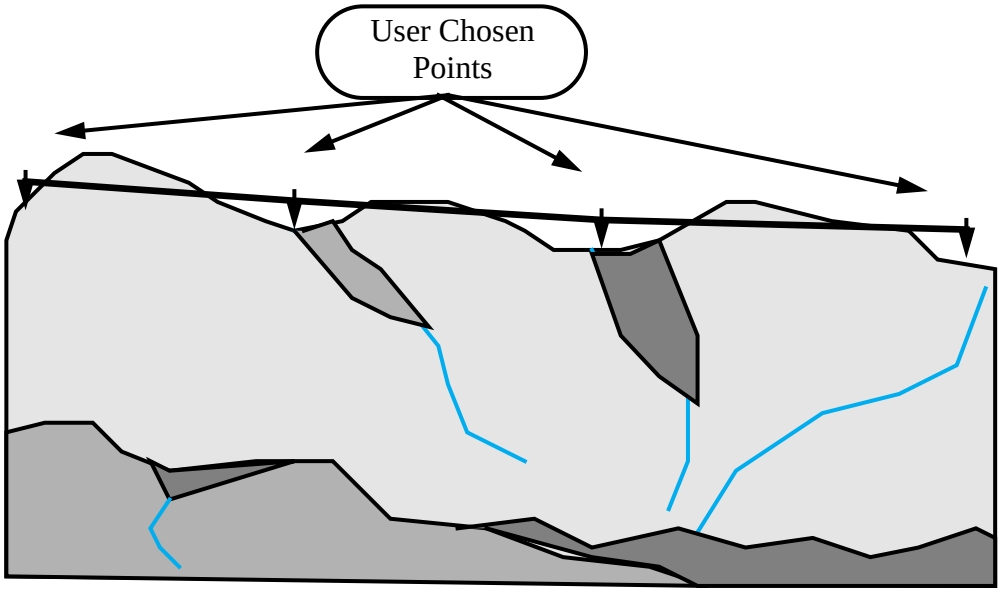


Figure 58: Calculating Line Strip Vertices Using Only `points`

The result of subdivision is that during actual line construction `points` is used to construct initial line segments. The angle between each set of two vertices in `points` is calculated to form the initial line segment. This line segment is then subdivided into smaller segments giving a set of additional vertices located *on* the initial line segment, between the two original points, at the correct elevation. Each of the vertices is added to

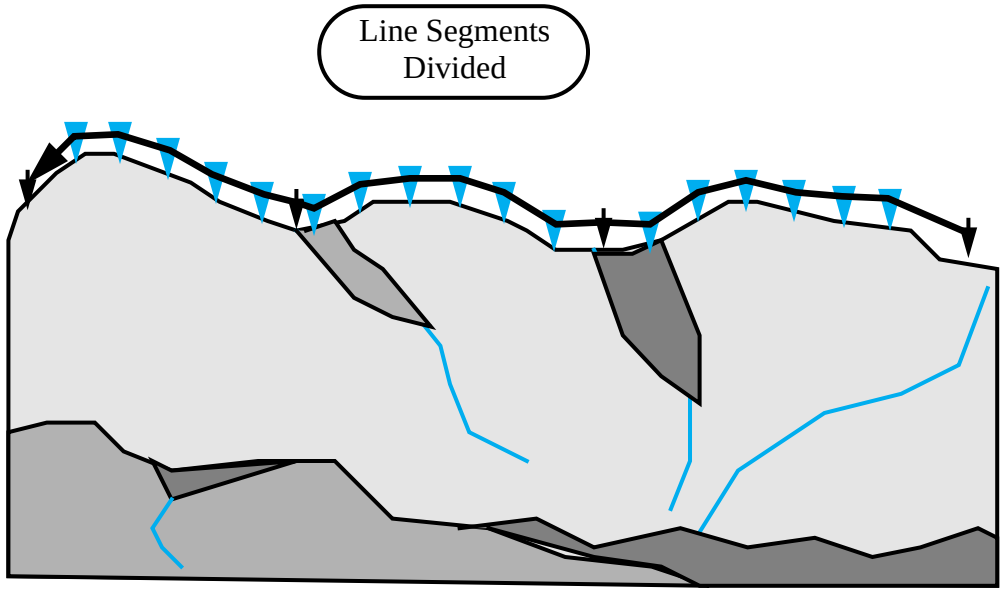


Figure 59: Calculating Line Strip Vertices Using Subdivision

an array of vertices stored at `coords` and the total number of vertices is stored at `length_vec`. `coords` and `length_vec` are then used for the actually construction of the pfGeoSet as shown in Figure 57. Lastly, if a linear measure is being constructed as a closed line loop a line segment is constructed from the last vertex in `points` to the first, the segment is subdivided as above and the vertices and number of vertices are added to `coords` and `length_vec`.

This completes the construction of the visible line strip; however, the transparent triangle strip used for picking must also be constructed. The triangle strip is oriented vertically and the pfGeoSet for the strip is created in the function `make_vertical`. The construction uses the points calculated in a manner similar to those used for the line strip; however, additional points are needed to provide all of the vertices for each of the triangles in the strip. Figure 60 depicts a “close-up” of a vertical triangle strip compared with the

line strip. In the line strip representation each "+" represents a vertex, and the line segment between the vertices are the sub-segments of the line strip. These were calculated during subdivision as described above. These vertices can be used directly to construct the line strip. These vertices are also used in the triangle strip; however, only in an intermediate step.

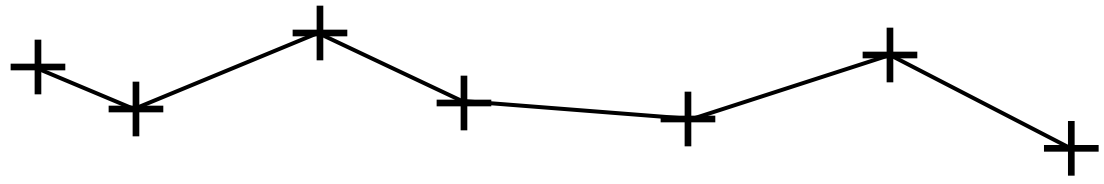
In the triangle strip representation the dotted center line and vertices represent the same subdivisions used in the line strip. To these vertices a distance, based on the desired thickness of the triangle strip, is added and subtracted vertically. The result is the actual vertices used in the triangle strip. These vertices are then put into `coords` and the cumulative number of vertices is placed in `length_vec`. Note, that for every vertex in the line strip, two vertices are required for the triangle strip.

With `coords` and `length_vec` acquired the pfGeoSet is constructed using PFGS_TRISTRIPS as the Performer primitive. Both pfGeoSets (line strip and triangle strip) are then put in pfGeodes and added to the parent group.

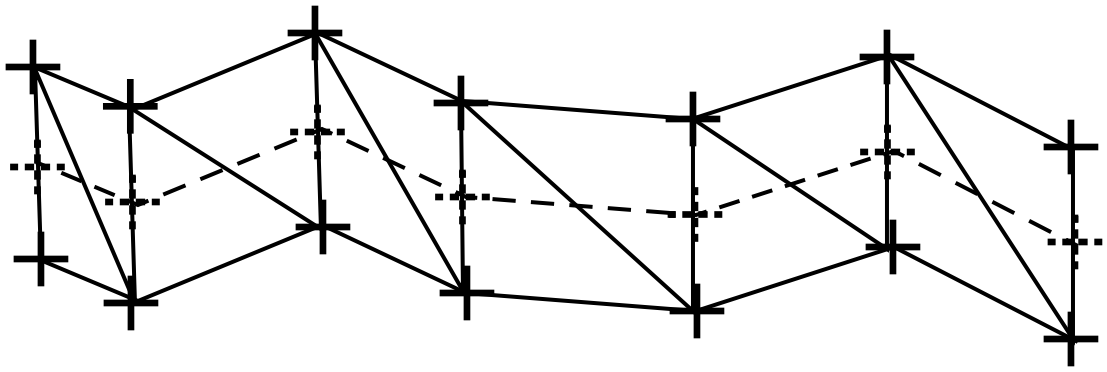
E. MINEFIELDS

The last version of `make_meas` is that used for minefields. The creation of minefields is very similar to the creation of closed line loops just presented with the difference being that minefields are constructed using *visible* triangle strips rather than line strips. The triangle strips are horizontal and appear as a "ribbon" placed over the terrain. The actual Performer structure for a minefield is the simplest of all the measures and since the measure uses triangle strips there are neither "picking" nor "highlighting" problems. The minefield Performer structure is shown in Figure 61.

The minefield is constructed exactly as the linear measure is constructed with the following exceptions. Only the triangle strip is constructed, it is visible, and the displacement used to calculate the triangle vertices is horizontal rather than vertical. This is slightly more difficult because the distance added to the center line is not simply up or down as was the case with the vertical strip. Rather, it is a vector component based on the



Line Strip Representation



Triangle Strip Representation

Figure 60: Vertical Triangle Strips

angular heading of the line segment being represented. This is solved with simple trigonometry and is not presented here. Once the vertices have been calculated the pfGeoSet, pfGeode and pfGroup are constructed using the same method as was used for the linear measures.

F. PICKING, DRAGGING AND DROPPING

While previously eluded to, the implementation of picking, dragging and dropping will now be examined. As was seen in the previous sections, many of the design considerations used in building the control measures were made with picking, dragging and

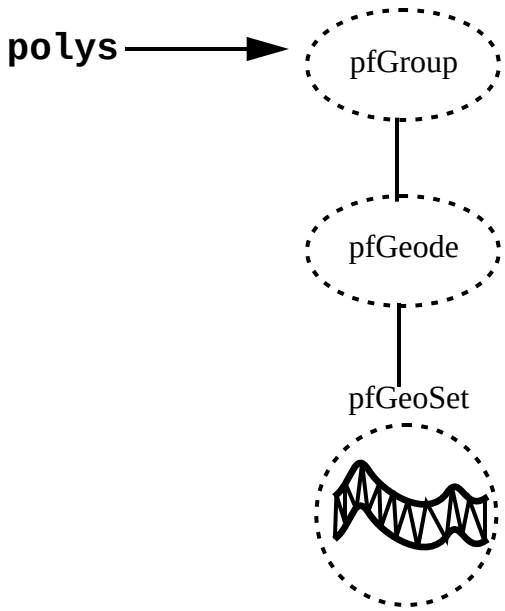


Figure 61: Minefield Performer Structure

dropping in mind. The resulting structures for the measures are quite effective and sufficient for creating intuitive manipulation of control measures.

1. Picking

A goal in designing the “picking” system was to use the functionality provided by Performer. Additionally, basic “picking” functionality was available in the Silicon Graphics program *Pickfly* which is released for unrestricted use. By extending this program, the “picking” functionality of the Sand Table was created. The functions used in “picking” are contained in picking.cc.

The basic flow of the picking function is as follows. The function prepares for a pick by initializing the “pick” to the Performer scene which is being picked from. Next, mouse

screen coordinates are normalized to world coordinates. The pick is then made and its results are returned in a variable of type pfHit. This pfHit variable is then queried for the pfNode from the scene graph which intersected the cursors normalized position. The pfHit can also be queried for the actual geometry coordinate of the point in the scene where the mouse intersected and the path taken in the Performer scene to arrive at the intersecting pfNode. Each of these queries is used in the Sand Table. A structure, Pick, is created to store many of the variables used during "picking". The structure presented in Figure 62 and Figure 63 is an extract of C++ code from `DoPick` which is the function which actually implements the pick outlined above (code not examined has been omitted for brevity).

```
typedef struct pickstruct{
      pfScene    *scene;
      pfChannel  **chan;
      pfNode     *picked;
      pfGroup    *group;
      pfVec3     point;
      pfPath     *path;
      char       *pathname;
      float      pickX;
      float      pickY;
      long       traverse;
}  Pick;
```

Figure 62: Pick Structure

The function `DoPick` takes a variable `P` of type Pick and mouse screen coordinates as input. The function then fills the pick structure as will be shown and returns a pointer to the node which was selected from the pick. Prior to calling the function, the Performer scene and channel against which the pick is being made are loaded into `P`. `pfNodePickSetup` initializes the pick mechanism with the scene. `pfuCalcNormalizedChanXY` normalizes the mouse screen coordinates to the channel being picked against and stores the normalized coordinates in `P->pickX` and `P->pickY`. These normalized coordinates are then used in `pfChanPick` to actually traverse the scene

```
pfNode *DoPick(Pick *P, long mousex, long mousey){
      long            pick_count = 0;
      static pfHit    **ipicked[10];

      pfNodePickSetup(P->scene);
      pfuCalcNormalizedChanXY(&P-pickX, &P->pickY,
                              *P->chan, mousex,mousey);
      pick_count = pfChanPick(*P->chan, P->traverse,
                              P->pickX, P->pickY, 0.0,
                              ipicked);

      pfQueryHit(*ipicked[0], PFQHIT_NODE, &P->picked);
      pfQueryHit(*ipicked[0], PFQHIT_POINT, &P->point);
      pfQueryHit(*ipicked[0], PFQHIT_PATH, &P->path);

...
}
```

Figure 63: DoPick

graph and determine if mouse hits have been made. If so, these hits are stored in `ipicked`. `ipicked` can then be queried to find what was actually picked with the mouse hit.

`ipicked` is queried for three values, `PFQHIT_NODE`, `PFQHIT_POINT` and `PFQHIT_PATH`. These correspond to the pfNode which was selected, the coordinate of the point on the scene geometry and the path taken to the node. These values are loaded in `P` and contain the information actually used by the Sand Table.

When a node is picked its path is examined to see if the pfGroup `L_po_meas` is included. If so, the selected node is a control measure because, recalling Figure 52, all control measures are added as children to the `L_po_meas` pfGroup. If a control measure has been hit, the parent group of the measure can be found by starting with the actual pfNode which *was* hit and following the parents back to the pfGroup at the top of that particular measure. The measure is then highlighted in a manner appropriate for that measure as discussed above achieving the “hot mouse” functionality desired. Further, since the parent pfGroup has been obtained, a pointer to that group can be used as a key to

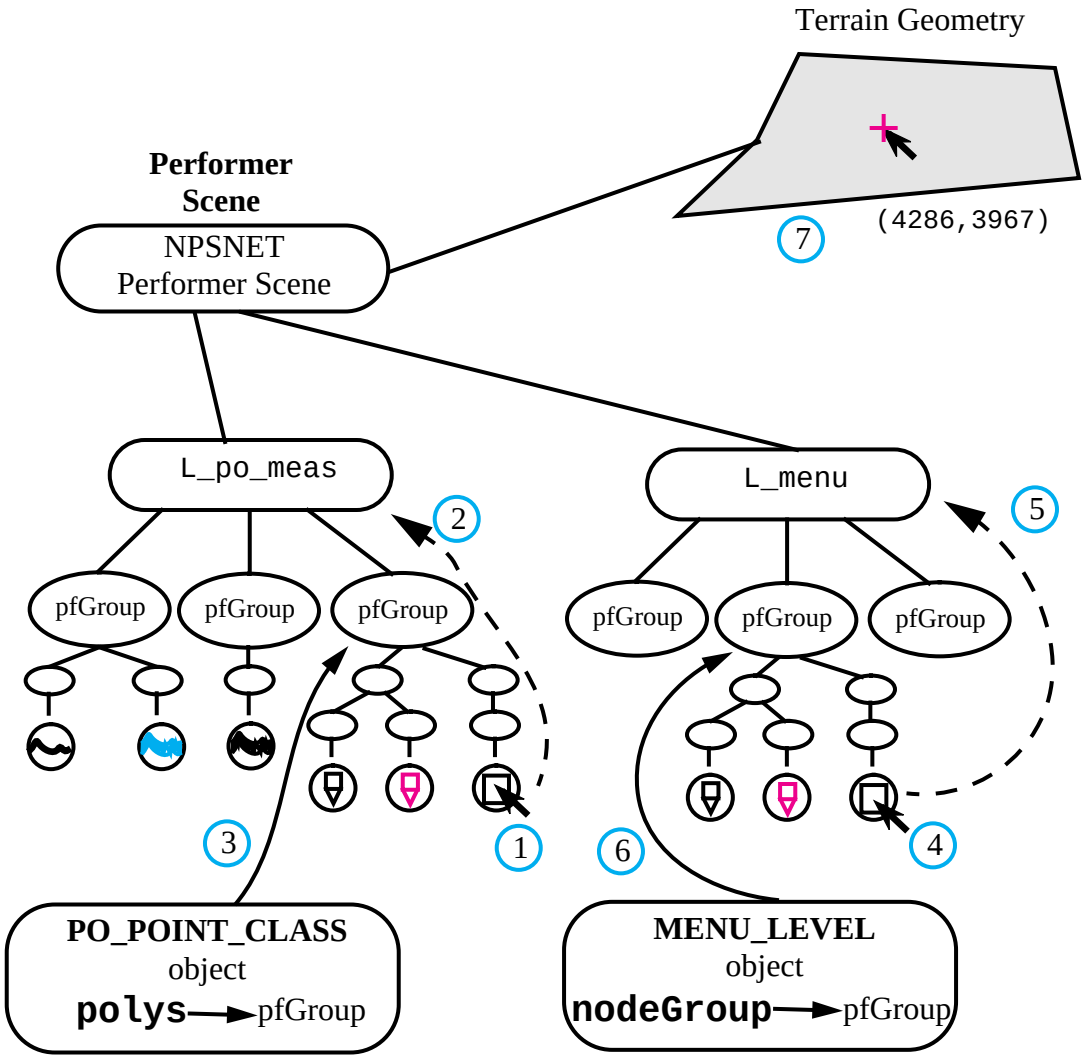
find the actual PO_MEASURES_CLASS object which represents the measure in the Sand Table database. The pointer to the group would be compared against the member `polys` in the PO_MEASURES_CLASS object. Thus we have achieved the capability of accessing a control measure in the database via picking from the three dimensional scene. This will serve as the basis for all manipulation of the control measures on the Sand Table.

The possibility also exists, that when the path is examined from a node which was picked, `L_po_meas` was not picked. This could be any object in the three dimensional scene. However, this version of the Sand Table is concerned with only two possible objects. The first, is a pfNode which is part of the cone tree menu. This possibility is handled in the same way in which control measures were handled in that the path is examined to find a parent. However, the path is not examined for `L_po_meas` rather, it is examined for the pfGroup `L_menu` which is the parent group for all Performer menu structures. The path within the Performer graph can also be followed back to a parent group which will be used for a callback mechanism within the menu system. The cone tree menu will be discussed further in Section H.

The other possible object, or actually Performer geometry, which is of interest to the Sand Table, is the terrain itself. The `PFQHIT_POINT` query returns the coordinates of the Performer geometry selected in a pfHit. Using this the Sand Table has the capability to select a point on the terrain and obtain its coordinates via mouse input from the three dimensional scene. This will be used during dragging and dropping, menu placement and placement of points used in the construction of lines and points.

A summary of the Sand Table “picking” mechanism of `DoPick` is shown in Figure 64. The MENU_LEVEL class shown in the figure has not yet been introduced; however, the figure shows how a graphical selection gives a pointer which is used by the menu system

Within po_funcs.cc is a function `HLPick` which actually calls `DoPick`. Recall, that `DoPick` loads a variable of type `Pick` with the query values of the pfHit. `HLPick` processes two types of picks from `DoPick`, those being control measures (children of `L_po_meas` in Figure 64) and terrain locations. The menu system has its own function,



Possible Picking Events Which May Occur. 1. Geometry from measure is "hit" with cursor. **2.** Queried path contains `L_po_meas`. **3.** Scene graph traced to parent group which is same group pointed to by `polys` in PO_MEASURES_CLASS object. **4.** Geometry from menu icon is "hit" with cursor. **5.** Queried path contains `L_menu`. **6.** Scene graph traced to parent group which is same group pointed to by `nodeGroup` in MENU_LEVEL object. **7.** Query returns intersection point of cursor on terrain.

Figure 64: Sand Table Picking

menuPick, which processes picked menu icons (children of L_menu in Figure 64). With these values and mouse button values the appropriate actions are taken.

The flow of HLPick resembles a finite state machine in that the actions taken depend not only on the current values of the selected group and mouse button values, but also prior values. The state of the function can be determined by the current values of the mouse buttons and the values of the variables pickedGroup, which is the parent pfGroup returned from DoPick, and mouseWasDown which was the value of the mouse button the last time through the function. Both of these variables have the value of zero prior to entry into HLPick. The flow is best described in a diagram which is presented in Figure 65 and described below.

DoPick is called and a variable of type Pick is returned. The pfGroup is extracted from Pick and checked to see if it is a control measure or terrain (logical OR). If not, the mouse button is checked. If depressed pickedGroup and mouseWasDown are checked. If pickedGroup != 0 and mouseWasDown = 1 then a measure is being dragged and the most recent mouse position is ignored. This would occur for example if a measure was being dragged and it was dragged across the menu icons. Appropriately, nothing would happen. If a measure is not being dragged and the mouse button is depressed, the pick is processed elsewhere, in the current version of the Sand Table this processing would occur in menuPick. If the mouse button is not depressed any "highlighting", which may have be on, is turned off and pickedGroup and mouseWasDown are both set to zero. This would occur if the mouse cursor was moved off of a measure and onto another object in the scene.

If the pfGroup selected was terrain or a control measure (logical OR) the mouse button is again checked. If depressed pickedGroup and mouseWasDown are checked. If pickedGroup != 0 and mouseWasDown = 1 then a measure is being dragged and the drag is processed. If pickedGroup = 0 or mouseWasDown = 0 the extracted pfGroup is examined to determine whether it is terrain or a control measure. If terrain, the coordinates of the point on the terrain are retrieved (also in Pick) and possibly used for building a line or point measure. This will be discussed with the cone tree menu system.

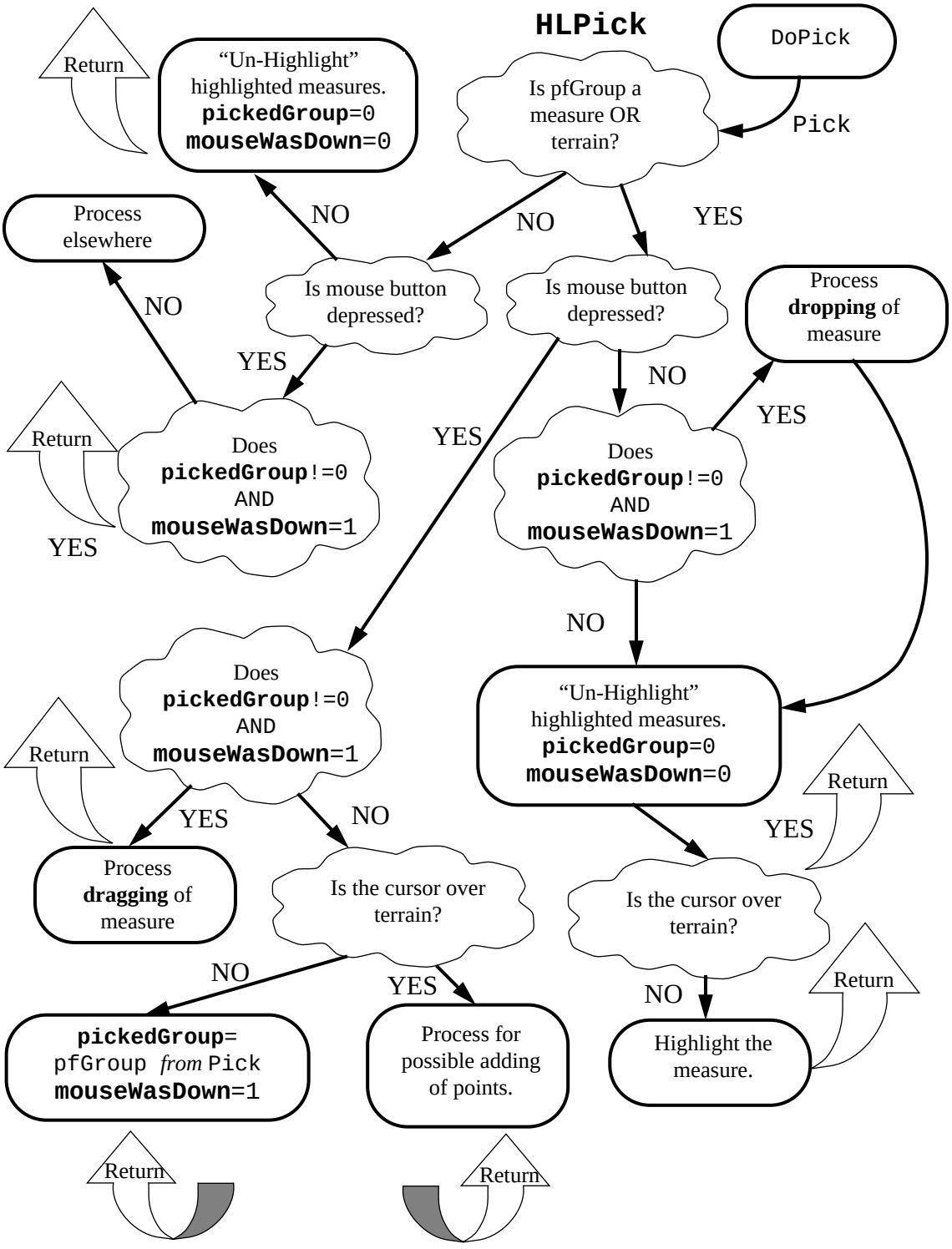


Figure 65: Flow of `HLPick`

If a measure, then the pick is the initial selection of a measure to be dragged. `pickedGroup` is set to the extracted pfGroup and `mouseWasDown` is set to one enabling the measure to be dragged the next time `HLPick` is called.

If the mouse button was not depressed and terrain or a measure were selected, `pickedGroup` and `mouseWasDown` are checked. If `pickedGroup != 0` and `mouseWasDown = 1` then a measure *was* being dragged and is being dropped. If not, then either terrain has been "hit" in which case any highlighted measures are "un-highlighted" or a measure is "hit" and highlighted.

2. Dragging and Dropping of Control Measures

With the "picking" capability described above the task of dragging and dropping control measures becomes very straightforward. Three polymorphic member functions within PO_MEASURES_CLASS (see Figure 21) are used in dragging and dropping and are called from `HLPick`. These functions are `movePO`, `dragUpdatePO` and `dragToNet`. The actual function for each derived class will be examined in the sub-sections below.

a. Dragging and Dropping of Points

When the conditions of Figure 65 have been met for dragging a point, the PO_POINT_CLASS member function `movePO` executes the drag. The function exploits the duplicate geometry of the point measure's Performer structure actually recouping some of the performance cost of the redundant structure. The function `movePO` is given in Figure 66. Referring to Figure 54 will aid in understanding `movePO`.

The function is called with the coordinates of the terrain at which the cursor is located (see Figure 62). `outToNet` is used as a flag to indicate whether or not the drag should be sent to the network. Both the highlighted and measure color version of the point geometry are selected with the `pfSwitchVal` statement so both will be rendered. The pfBillboard which contains the highlighted version is then translated to the point at which the cursor is located on the terrain. The effect is that when a point is being dragged, a point icon of the measure color remains in the original position of the point on the terrain, while

```
void PO_POINT_CLASS::movePO(float xpos, float ypos, ushort
                            outToNet){
      pfSwitch*       ptSwitch;
      pfBillboard*    board;
      pfVec3          tempPoints;

      tempPoints[X] = xpos;
      tempPoints[Y] = ypos;
      tempPoints[Z] = gnd_level2(tempPoints);

      ptSwitch = (pfSwitch*) pfGetChild(polys, 0);
      pfSwitchVal (ptSwitch, PFSWITCH_ON);
      board = (pfBillboard*) pfGetChild(ptSwitch, 1);

      pfBboardPos(board, 0, tempPoints);

      if (outToNet) dragToNet(xpos, ypos);
}
```

Figure 66: `movePO`

a highlighted point icon is moved dynamically to a new point on the terrain. This is updated each time through the application process and the point is moved across the terrain in real-time. Note, that the actual member values of the PO_POINT_OBJECT which actually determine the state of the object are *not* changed during the drag.

If the drag is being sent to the network `dragToNet` is called. `dragToNet` is basically like `sendPDU` (see page 52) with the exception that `updatePDU` is not called and the only part of the `SP_DescribeObjectVariant` PDU which is changed is the location of the point. A Sand Table or ModSAF station receiving these PDUs treat them as normal with the function `update` (see page 51). The result is that the point measure will move over terrain on the other simulator as it is being dragged on the local simulator, depending on network traffic, in real time.

When the mouse button is released after dragging a point the function `dragUpdatePO` is called. The function is given in Figure 67. The function is very similar

```
void PO_POINT_CLASS::dragUpdatePO(float xpos, float ypos,
                                  ushort outToNet){
      pfSwitch*       ptSwitch;
      pfBillboard*    board, *board2;
      pfDCS           DCS;

      points[X] = xpos;
      points[Y] = ypos;
      points[Z] = gnd_level2(points[0]);

      ptSwitch = (pfSwitch*) pfGetChild(polys, 0);
      pfSwitchVal (ptSwitch, 0);
      board = (pfBillboard*) pfGetChild(ptSwitch, 0);
      board2 = (pfBillboard*) pfGetChild(ptSwitch, 1);

      DCS = (pfDCS*)pfGetChild(polys, 1);
      pfDCSTrans(DCS, points[0][X],points[0][Y],
                 points[0][Z]);

      pfBboardPos(board, 0, points[0]);
      pfBboardPos(board2, 0, points[0]);

      if (outToNet) {
            updatePDU();
            sendPDU();
      }
}
```

Figure 67: `dragUpdatePO`

to `movePO` except that the points read from the terrain are actually put into the member `points` which actually changes the state of the PO_POINT_CLASS object. Next, the `pfSwitchVal` sets the switch so that only the measure colored geometry is rendered. The billboard locations are set to the location of `points` as is the DCS which makes both representations of the point geometry and the backboard quad coincident on the terrain. Lastly, if the drop is being sent to the network, `updatePDU` updates the member `pdu` and the `SP_DescribeObjectVariant` PDU is sent to the network.

b. Dragging and Dropping of Lines and Minefields

The dragging and dropping of lines and minefields use the same methods so only lines will be discussed. The dragging of lines is much more difficult than points for two reasons. The first is the difficulty of moving lines in real time. This is due to the fact that constructing a line which follows the contour of the terrain must be done at every new position. The actual construction takes a relatively large amount of time which would be a noticed latency in the moving of a line. The second difficulty is that lines do not possess an easily manipulable pfDCS nor pfBillboard as points do.

The first problem is solved by using a lower resolution representation of the line when it is being dragged. Recall (see Figure 59) that when the geometry for a line is created, a subdivision is used to create "terrain following". In order to create a lower resolution line, the size of each segment is simply increased, thereby decreasing the number needed. If the line has a low enough resolution, the computation is minimal and can be completed in real time. The trade off, which is the possibility that the line may intersect terrain, was already eluded to in Figure 58. This was deemed acceptable since the real time dragging was considered more important and the low resolution representation of the line is only used during dragging. This solution is implemented by using a function called `makeQuickLine`. `makeQuickLine` creates a low resolution line which is added to the pfGroup `polys`. As soon as the line is dropped the higher resolution line is again used and the line created by `makeQuickLine` is removed from the scene.

The second problem is solved by saving a reference point when the line is first selected to be dragged. On subsequent calls to `movePO`, the location of the cursor on the terrain is compared to the reference point and differences in the x and y coordinates are calculated. These difference are then applied to each of the pfVec3s in `points` to give the new point locations. These locations are then used by `makeQuickLine` to construct a low resolution line representation which is displaced by the correct amount. This is demonstrated in Figure 68. Note that the high and low resolution lines are depicted in the x-y plane which is inaccurate. The resolution is actually in the *vertical* plane. However,

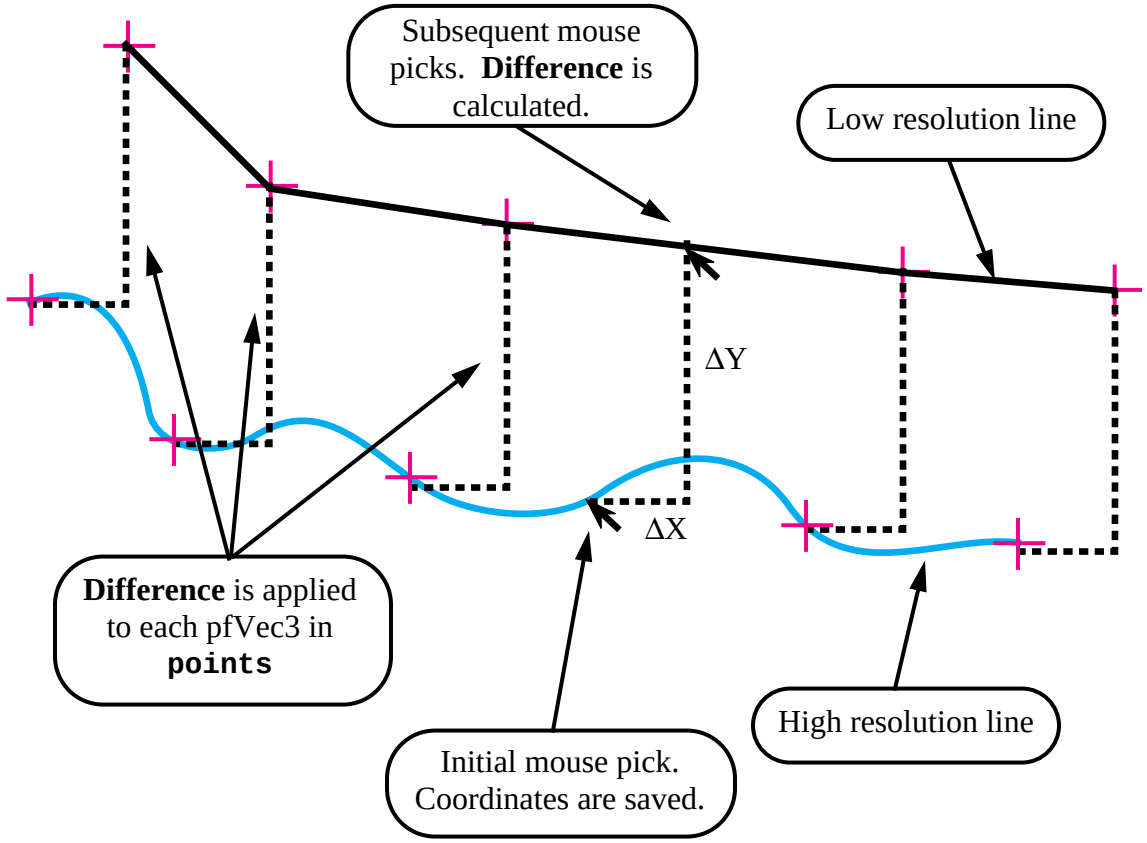


Figure 68: Dragging a Line

this is a limitation of the figure and the point is still made. The x and y differences *are* depicted correctly.

Having described the components of `movePO` the operation of the function will be summarized. The function is called with coordinates from the terrain. The coordinates are saved as a reference point. On subsequent calls the coordinates which are passed to the function are compared to the reference point. A difference is calculated and is applied to `points` and `makeQuickLine` creates the low resolution line. Any old low resolution lines (from previous invocations of `movePO`) are removed from the scene. The new low resolution line is added to the scene and if required the drag is sent to the network with

`dragToNet`. Again note, that no changes are made to the member functions of the PO_LINE_CLASS object in the drag operation.

`dragToNet` also presents difficulties in the dragging of lines. This is due to the function `update` which, upon receiving a `SP_DescribeObjectVariant` PDU for the line being dragged, builds a high resolution representation of the line. To solve this problem an unused field in the `SP_LineClass` variant of the `SP_DescribeObjectVariant` PDU is used (see Figure 11). The field is named `_unused_10` and is not employed by ModSAF. (The field is named `_unused_13` for the minefield variant. (see Figure 15) The Sand Table, however, uses the field to flag whether or not a control measure is being dragged. The function `update` then checks the field and, if the measure is being dragged, constructs the measure using `makeQuickLine`. Thus, both the local and remote simulator will render the line or minefield being moved in real time.

Lastly, when the measure is dropped, the low resolution representation of the line is removed from the Performer scene graph. The difference calculated from the reference point is actually used to modify the member `points` and a high resolution representation is created and assigned to `polys`. `updatePDU` is called which in the case of the line and minefield also resets the flag which indicates the measure is being dragged. `sendPDU` then sends the modified `SP_DescribeObjectVariant` PDU and all Sand Tables update the position of the measure with a high resolution representation.

G. CONE TREE MENU SYSTEM

The final topic covered in this chapter is the cone tree menu system. This system was seen in Chapter IV in the context of building points and lines on the Sand Table. This section is a more detailed examination of the menus and examines their actual design and functionality. The menu is depicted below in Figure 69.

The menu is designed to be placed on terrain at any location desired by the user. The menu is intended to be an “area of interest” tool meaning that it can be placed in the *area*

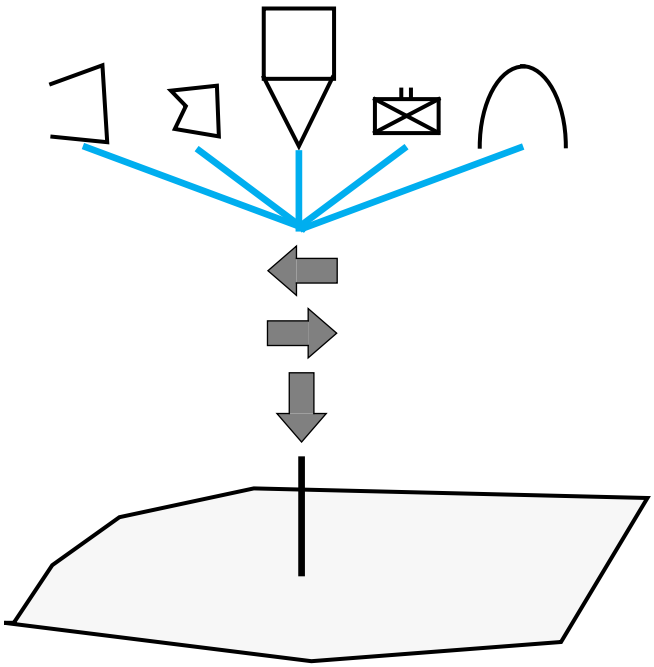


Figure 69: Cone Tree Menu

where the user is doing work on the Sand Table. This can be contrasted to the classic window style menu system in which the user must take his attention out of the scene, manipulate the menu and then return to the scene. The menu can also be dragged over the terrain or collapsed from view offering the user great flexibility to quickly position the menu where it is useful, yet not obscuring more important parts of the Sand Table.

The menu consists of two major components which are interconnected. These components are the graphical representation of the menu and the callback mechanism for the icons. The menu consists of the class MENU_LEVEL which is defined in menu.cc. The menu itself is literally a depiction of the Performer graph itself. By examining the prototype for MENU_LEVEL along with a “dissected” cone tree menu the close correlation is quickly seen and the menu can be understood more quickly. The prototype is given in Figure 70 and the “dissected” menu is presented in Figure 69.

```
class MENU_LEVEL{
public:
      pfDCS*     root;
      pfGroup*   nextLevelSwitch;
      pfGroup*   nodeGroup;
      pfDCS*     rotateDCS;
      pfGroup*   edgesGroup;
      pfGroup*   downGroup;
      pfGroup*   rightGroup;
      pfGroup*   leftGroup;

      pfVec3*    points;

      int numChildren;
      int childNumber;

      MENU_LEVEL*     nextLevel[MAX_CHILDREN];
      MENU_LEVEL*     previousLevel;

      float      cumRotation;

      void       simSwitch(int);
      MENU_LEVEL(pfGroup *node,
                 void(*function)(MENU_LEVEL*));
      MENU_LEVEL(pfGroup *node,
                 void(*function)(MENU_LEVEL*), void*);
      ~MENU_LEVEL();

      void addMenuChild(MENU_LEVEL* child);

};
```

Figure 70: MENU_LEVEL Class

The diagram shows the structure for *one* level in a cone tree menu. The actual Performer nodes are shown as well as their corresponding member names. When examining the Performer structure, the diagram is actually upside down, with the members at the bottom of the diagram being the parents of those nodes located higher in the diagram. At the base of each menu level is that level's `root` which is a pfDCS. In the case of the lowest menu level, the `root` serves to position and move the menu on the virtual terrain.

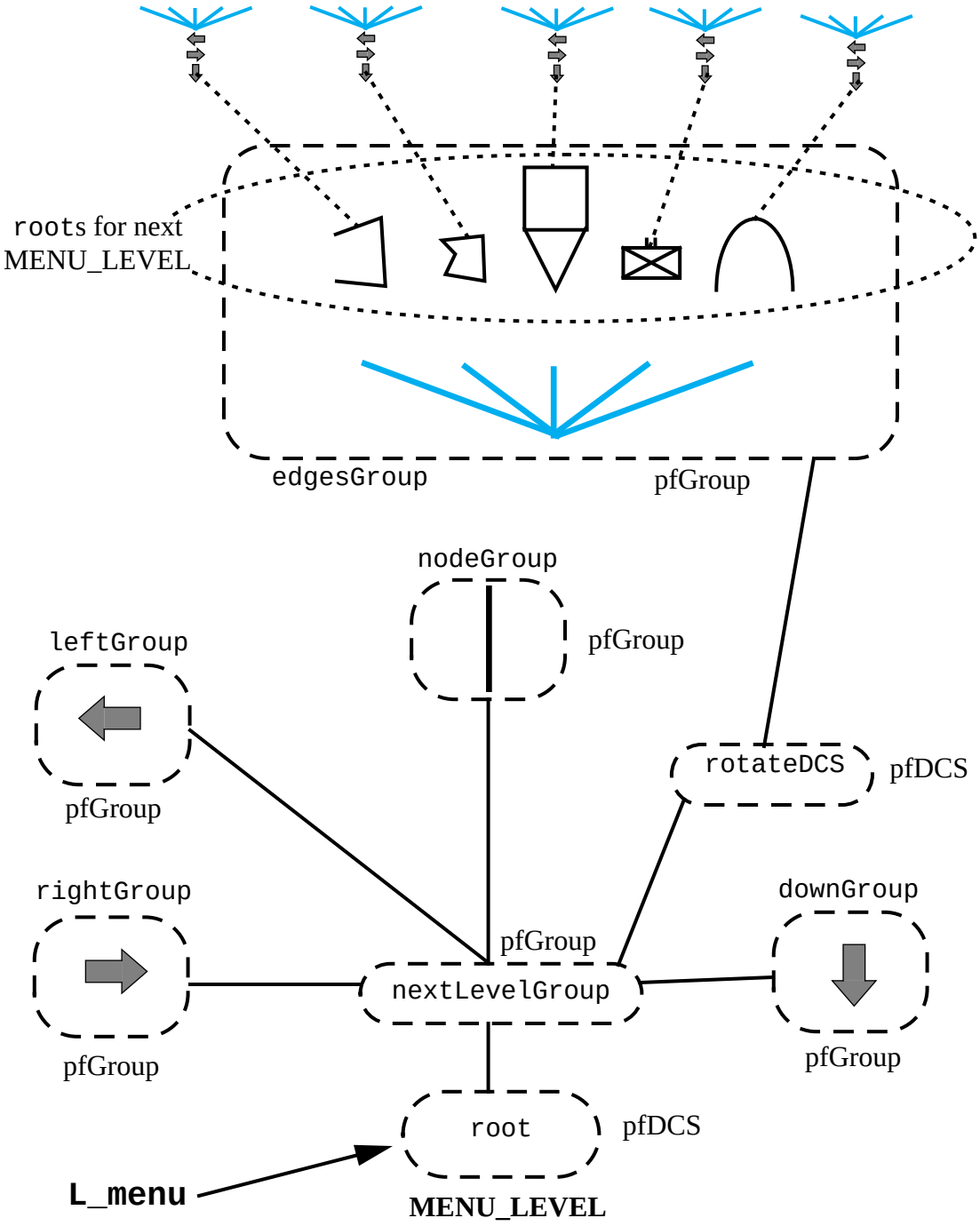


Figure 71: Dissected Cone Tree Menu

By translating this DCS, the menu can be positioned at any coordinate. pfDCSs in higher menu levels serve to place the root at the correct *relative* position of the cone structure. The lowest level root is also the node which is attached to `L_menu` which is the base of the menu which is attached to the rest of the NPSNET Performer scene graph (see Figure 64).

`nextLevelGroup` is the next node in the menu tree. `nextLevelGroup` is also the result of a Performer limitation. The desired functionality of the node was to control which of the children groups above `nextLevelGroup` would be displayed. This functionality seemed perfectly suited to a pfSwitch node which allow the selective choosing of nodes to be rendered. The difficulty arose with Performer's inability to "pick" through a pfSwitch during the graph traversal, the exact ability which is needed in the menu. The icons must be able to be queried with a pfHit so they can be located and the correct menu action can be carried out.

The solution was `nextLevelGroup` which is not a pfSwitch; rather, it is an ordinary pfGroup which is used to *simulate* a switch. This is accomplished with the function `simSwitch` which uses the Performer functions pfAddChild, pfGetChild and pfRemoveChild to selectively add and remove children from the `nextLevelGroup` pfGroup. The function is called with an integer parameter and can display all of its children, none of its children or selected children.

The children added to `nextLevelGroup` are `leftGroup`, `rightGroup`, `downGroup`, `nodeGroup` and `rotateDCS`. `leftGroup`, `rightGroup`, `downGroup` and `nodeGroup` each have a "pickable" icon associated with them. `leftGroup` and `rightGroup` rotate the next higher cone tree level. `downGroup` collapses the next higher cone tree level. `nodeGroup` is the actual icon for the chosen level. When selected it may expand the menu by one level (if not the terminator) and will execute a function appropriate to the desired functionality of the icon. The graphical structure of the icons is the same as for point control measures (see Figure 54) and is constructed in the same manner. However, different functions are used to actually create the structures. These functions are `makePickableTriStrip` and `makePickableLnStrip` which, as the names imply,

make a triangle strip and a line strip which can be “picked” and “highlighted” with the cursor using the structure of Figure 54. Both functions read vertices from files as did `make_meas` (see page 95). The functions are contained in the file menu_funcs.cc.

In addition to the graphical structure of the icons, each icon has an associated callback. These callbacks give the menu all of its functionality. The callbacks are implemented by using pfUserData. pfUserData is a Performer function which allows the programmer to associate a pointer to data, with a Performer node. In the case of the menu system, a pointer to a call back structure is assigned to the parent pfGroup of an icon’s graphical structure. The callback structure used in the Sand Table is depicted in Figure 73 and the function

```
struct callBack{
      void            (*function)(MENU_LEVEL*);
      MENU_LEVEL*     nodeLevel;
      void*           userData;
}
```

Figure 72: Callback Structure

which assigns callbacks to nodes,`assignCB` , is given in Figure 72. This function is contained in po_funcs.cc.

The callback structure contains a pointer to a function, `function`, which takes a MENU_LEVEL pointer as a parameter. This is the actual function which will be called when the icon is selected. The MENU_LEVEL pointer is passed to give access to the menu structure for menu expansion or menu collapse. `nodeLevel` is a pointer to that actual menu level. Lastly, `userData` can contain a pointer to any piece of data which can then be used by `function`. The allows maximum flexibility in what action the callback actually executes. `assignCB` simply allocates storage for a new callback and loads the callback with the values sent by the parameters. Lastly, using pfUserData, the pointer to

```
void assignCB(pfGroup* node, void (*function)(MENU_LEVEL*),
              MENU_LEVEL* level, void* userData){

       callBack*  groupCB;

       groupCB = (callBack*)pfMalloc(sizeof(callBack),
                  pfGetSharedArena() );

       groupCB->function = function;
       groupCB->nodeLevel = level;
       groupCB->userData = userData;

       pfUserData(node, groupCB);
}
```

Figure 73: `assignCB`

the callback is associated with the actual Performer node which would be graphically picked.

Callback operation can now be summarized. First, a pickable piece of the menu is selected with the cursor. The path is queried to see if it contains (see Figure 64) `L_menu`. If so the Performer path is traced to the parent pfGroup of the icon structure. From this pfGroup a pointer to the callback structure is extracted. From the callback structure the function to be executed uses the pointer to the applicable menu level for menu manipulation and user data for any other actions carried out by the function. This "picking" and callback functionality is executed from `menuPick` which was mentioned during the examination of picking.

Continuing with the structure of the menu, `rotateDCS` will be examined. `rotateDCS` enable the cone branches and the icons located on the end of them to rotate like a "lazy susan". This allows the user to spin the varying levels of the menu to give an uncluttered view of the desired icons. `rotateDCS`'s one child is `edgesGroup` which contains the graphical branches as well as the `root` nodes from the higher menu levels. Children are added to the menu using the member function `addMenuChild`.

When a menu level is passed to `addMenuChild` the member `numChildren` is examined to determine how many children have been added to that level. From the number of children the relative position for each child icon is calculated. Using `root`, each child menu level is translated to its appropriate location for that level. Lastly, two line strips, one white and one highlighted, are constructed to actually represent the branch. Both the branches and the child icons are then put into the `edgesGroup` pfGroup of the parent menu level.

Having described the components of the menu, its construction can now be described in full. Note their are two constructors for the MENU_LEVEL class. The first two parameters are the same for each constructor. The first parameter is the pfGroup at the base of the MENU_LEVEL being created, which will be assigned to `nodeGroup` in the constructor. This pfGroup is created with `makePickableTriStrip` or `makePickableLnStrip` prior to the constructor. The second parameter is a pointer to the callback function for the `nodeGroup` icon. The third parameter, only present in the second constructor, is used to pass any user data which is used by the `nodeGroup`'s callback.

Within the constructor the `leftGroup` and `rightGroup` icons are made with `makePickableTriStrip` and are assigned callbacks which rotate `rotateDCS`. `downGroup`'s icon is similarly created and a callback which collapses the menu is assigned to the icon. `edgesGroup` is created and is added as a child of `rotateDCS`. `edgesGroup` is an empty pfGroup at this time since no children have been added. The pfGroups are next added to `nextLevelGroup`. Lastly, `simSwitch` is called so that only `nodeGroup` is rendered.

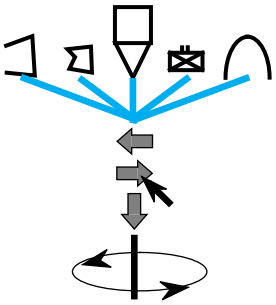
The entire menu is created only once and its subsequent display is controlled by manipulating `nextLevelGroup`. When the menu is first initialized the entire structure is built. The procedure in the proceeding paragraphs is followed for each menu level and `addMenuChild` is used to add child menu levels to parent levels. When the lowest level of the menu is created `simSwitch` is used to "turn on" all of the groups which are children of `nextLevelGroup` which results in the menu shown in Figure 69. Note, that the

children menu levels display only their `nodeGroup`. At this time if the `leftGroup` or `rightGroup` icons are selected they will rotate `rotateDCS` and maintain the amount of rotation in the MENU_LEVEL member `cumRotation`. If `downGroup` is selected, `simSwitch` removes all groups from `nextLevelGroup` of the lowest menu level.

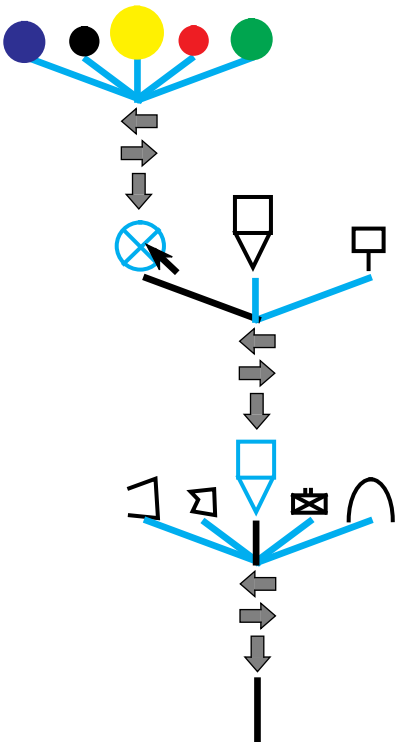
If a child icon is selected, the appropriate callback is called and the menu is expanded by calling `simSwitch` to "turn on" all of the groups in the member `nextLevelGroup` for the child menu level. The chosen icon is highlighted in the same manner point measures are and the highlighted color of the branch is chosen. Again note, that the newly expanded menu level only displays its `nodeGroup`. An example of the concepts described above is shown in Figure 74 and Figure 75.

H. SUMMARY

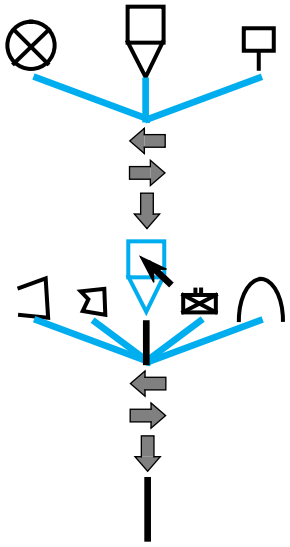
This chapter has presented the design and implementation of the graphical structures used in the Sand Table. An overview of the desired functionality was given. The actual Performer structure for each of the control measures was given as well as discussion as to why particular structures were chosen. The processes used to "pick" objects and terrain from the Sand Table scene were also discussed. The methods used in the dragging and dropping of measures were examined. Lastly, the actual structure and implementation of the cone tree menu structure was specified.



1a. Menu Placement.
Menu built. Lowest level `nextLevelGroup` displays all children. Next level `nextLevelGroup` displays only `nodeGroup`. **1b. `rightGroup` Icon Chosen.** `rotateDCS` rotated.

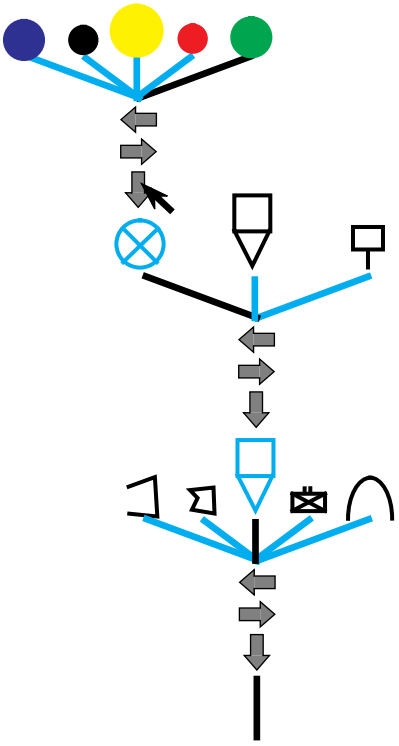


3. Coordinating Point Icon Selected.
Same as **2.**

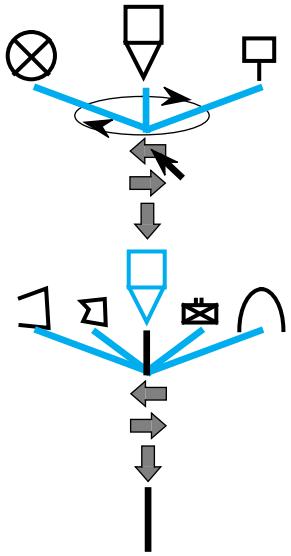


2. Build Point Icon Selected.
Icon is highlighted as point measure, branch highlight color is switched. Child's `nextLevelGroup` displays all children.

Figure 74: Actions in Menu Expansion



4. `downGroup` Icon Chosen. `nextLevelGroup` removes all children except `nodeGroup`. Icon normal color restored, branch no longer highlighted Resulting



5. `leftGroup` Icon Selected `rotateDCS` rotated

Figure 75: Actions in Menu Expansion(Continued)

VI. WEAPON FLIGHT PATH VISUALIZER

A. INTRODUCTION

1. Motivation

The purpose of the Sand Table in the virtual environment is to visualize aspects of battle planning which have not been available in previous paradigms. Thus far, the effort presented has focused on the rendering, management and manipulation of control measures on the virtual terrain. However, the concept of *abstract visualization* can additionally be applied to other areas of interest in battlefield planning. One of these areas is that of visualizing weapons' trajectories. Of key interest to the military planner are the spatial relationships of weapons' fires over the terrain. Put more simply, the commander would benefit by seeing where in space bullets, artillery and mortar rounds are flying.

This is of crucial importance to a military commander on the modern battlefield where there is increasing emphasis on maneuver warfare. Simply put, in a modern battle many types of weapons will be used in concert to create of synergy of weapons effectiveness. For example, attack helicopters may be used in close proximity to artillery firing, both of which are supporting an infantry advance. The benefit is the aforementioned synergistic effect. Conversely, the danger in employing weapons in such a manner is the possibility that friendly units could come under fire from weapons of friendly forces. Extending the above example, the attack helicopters could fly through the weapons trajectories of the artillery creating disastrous effects.

The solution to these dangers has always been meticulous coordination between units to ensure deconfliction between units. However, this process has been conducted by using maps and overlays with weapons ranges and directions of fire. Additionally, altitudes of weapons trajectories are well understood; however, when applied in the area of weapons' deconfliction the altitudes of weapons are given as altitude "blocks" for the planner to

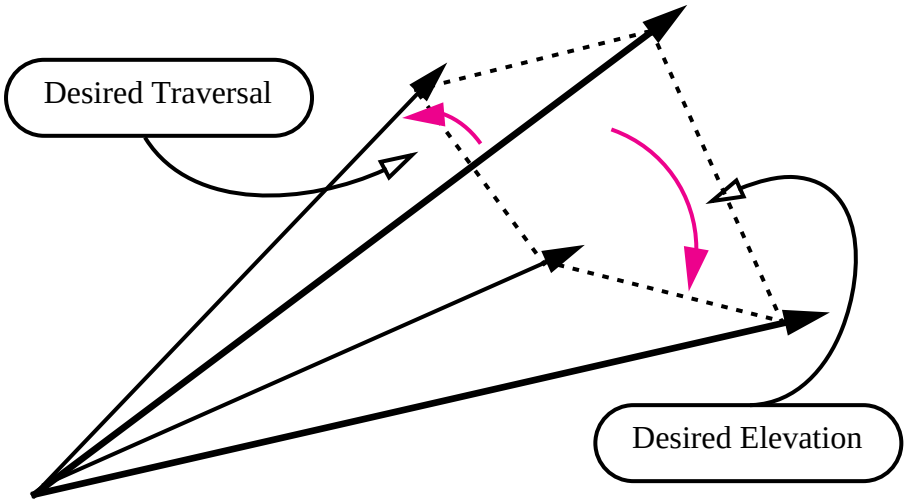
correctly interpret. The true three dimensional aspect of a weapon being fired is lost and left to the ability of planners and participants to correctly visualize the weapons. Decisions concerning "close calls" usually err towards a more safe choice to avoid possible conflict; however, when a safer choice is made the synergistic effects of closely coordinated weapons' fires may be reduced.

The problem presents itself as an excellent candidate for visualization on the Sand Table in the virtual environment. Weapons flight paths can be visualized over the actual terrain the weapons are being fired. Much of the interpretation concerning the actual flight paths is eliminated giving planners a common frame of reference. While physics prevents knowing exactly where every round fired will go, a physically based, graphically visualized representation of weapons' fires can beneficial in the coordination between friendly units. Returning to the above example, if attack helicopters could actually "see" where artillery was firing, the helicopter crews could successfully avoid friendly fire, while at the same time conducting their own missions in very close proximity to the artillery trajectories, thereby, allowing a more concentrated attack on a potential target. Additionally, such a visualization when applied to enemy weapons, can give a commander insight on vulnerabilities to his units which previously had to be interpreted from enemy weapons positions. This previous interpretation is dependant on the commanders skill and experience.

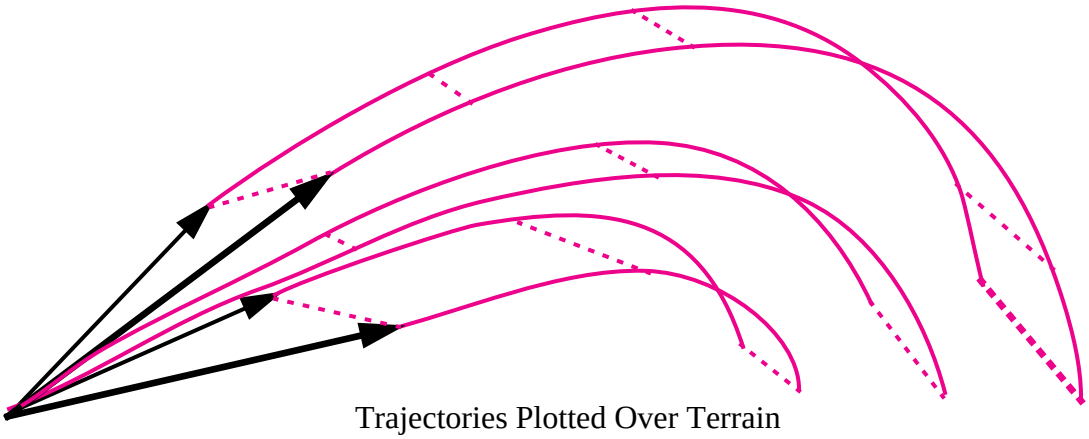
2. Approach

The approach taken was to implement a weapon's flight path visualizer in NPSNET. What this entails is the user being able to select a specific type of weapon and place it on the virtual terrain. The user will then select an area of fire where the weapon will be used. For example, an artillery piece will be given a range of bearings over which the weapon will traverse and a range of elevations over which the weapon will be elevated when fired. Once the area of coverage is selected, the user will then select any other parameters which will have an effect on the path of the projectile, such as gun propellant or charge strength.

The trajectories of the weapon will then be visualized over the NPSNET terrain. Figure 76



User Chooses Area of Coverage



Trajectories Plotted Over Terrain

Figure 76: Weapon's Trajectory Visualization

illustrates the desired visualization.

B. BALLISTICS BOX

In order to accomplish the desired functionality, a Performer structure very similar to the cone tree menu is constructed. The structure will be referred to as the *ballistics box*. The functionality of the ballistics box will first be discussed followed by a description of the implementation.

1. Ballistics Box Functionality

Recalling the cone tree menu system, a base level icon depicting a parabola was present. When this icon is chosen the menu expands to the level shown in Figure 77. Each

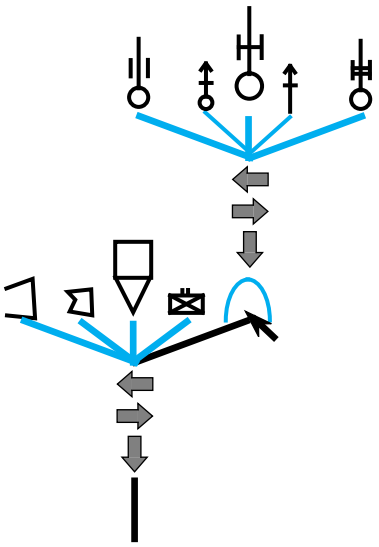


Figure 77: Expansion of Menu to Terminator

of the icons at the expanded level represent different weapons systems which are shown in Figure 78. A weapon icon is selected and expanded to the terminator. When the terminator is selected the cone tree menu disappears and is replaced with the ballistics box shown in Figure 79. The operation of the ballistics box is very similar to that of the cone tree menu. From the ballistics box the user chooses the parameters of the selected weapon and selects

Figure 78: Weapon Symbols

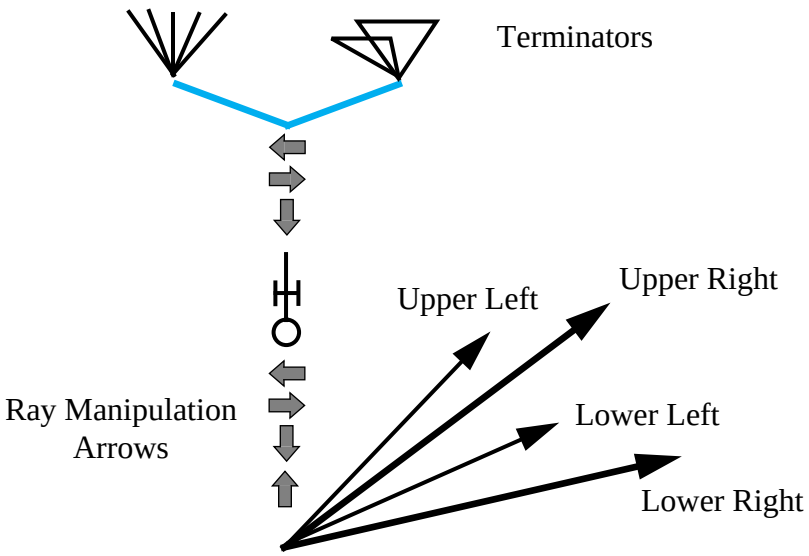


Figure 79: BALL_BOX Menu

one of the two top icons. In the case of the ballistics box there is no check mark terminator. Rather, the “fan” and “line fan” icons serve as terminators. When one of these is selected, the ballistics box disappears and the actual trajectory paths are displayed over the terrain as either three dimensional “fans” or a set of lines which represent the possible projectile flight paths.

The “rays” of the ballistics box are themselves selectable and can be rotated. The lower set of arrow icons rotate the selected rays. Although only one “ray” is selected at a time, its matching pair is also rotated. For example, if the lower right “ray” were selected and the left arrow icon was selected, both the lower and upper right rays would rotate. Similarly, if the up arrow were chosen both the lower right and lower left “rays” would be elevated. The limits of rotation are the actual degree limitations for the chosen weapon. Operation of the ballistics box are summarized in Figure 77 and Figure 77.

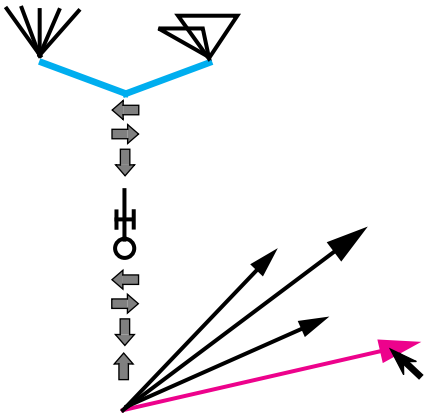
2. Trajectory Visualization

As was described above, either a fan or lines are used to represent the possible flight paths of the weapons projectile. The actual flight paths are calculated using a physically based fourth order Runge-Kutta method. Once the user selects the terminator, the paths at various incremental angles of elevation and traversal are calculated. Input into the physical model are the elevation angle of firing, the angular heading of the weapon and the muzzle velocity of the weapon. The points obtained from the Runge-Kutta are then either made into line strips or triangle strips depending on the user preference. The current physical model of the trajectories is not very accurate; however, the emphasis was on the visualization and a more accurate model can easily be employed at some later time.

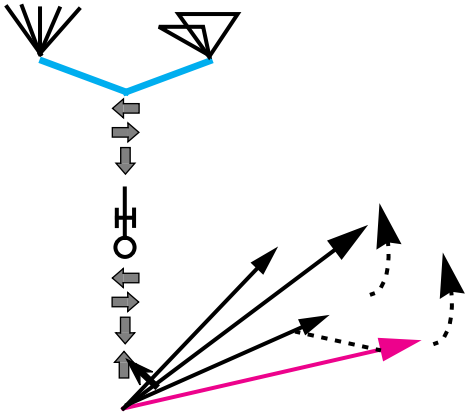
Line strips are constructed by using the points provided by the Runge-Kutta directly in the construction of a line strip primitive pfGeoSet. The construction of the fans is accomplished using triangle strips. The trajectory is displayed for several angular elevations. At each elevation the left and right traversal limits are used as vertices in the construction of the triangle strip. This is shown in an overhead two dimensional view of a fan in Figure 82.

3. Ballistics Box Design and Implementation

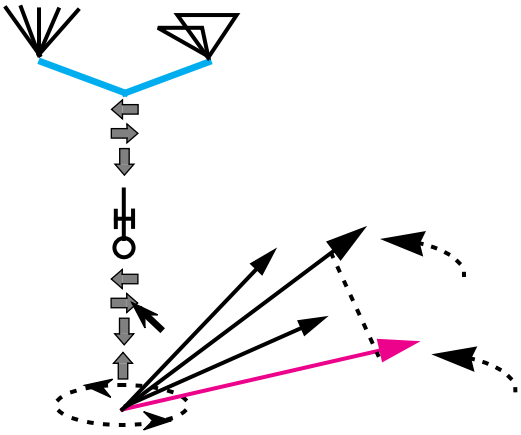
Having discussed the actual functionality and typical operation of the ballistics box, the actual design and implementation of the ballistics box will now be examined. The ballistics box is implemented using the class BALL_BOX which is contained in the file



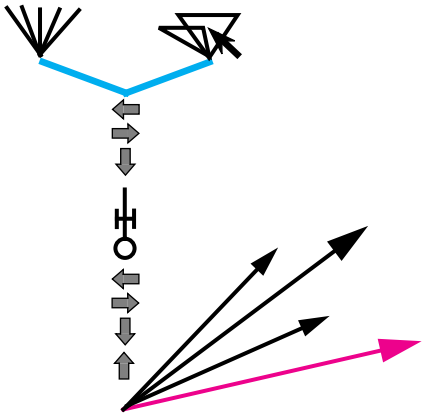
1. Ray Selected.
Selected ray is now capable of being rotated or elevated around center.



3. Up Arrow Selected.
Lower Right and Lower Left "rays" elevated. User positions as desired.

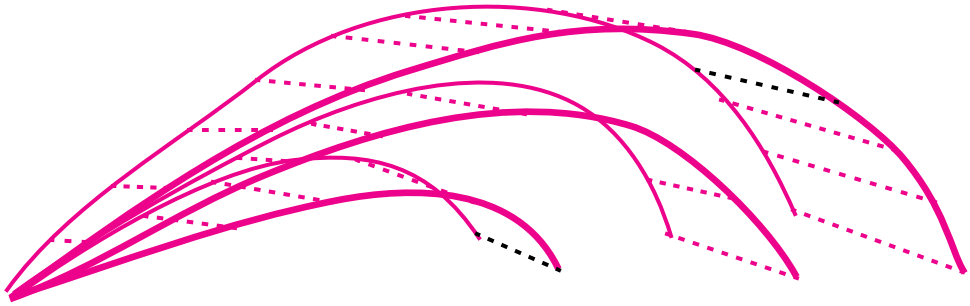


2. Right Arrow is Selected.
Lower Right and Upper Right "rays" are rotated around center. User positions as desired.

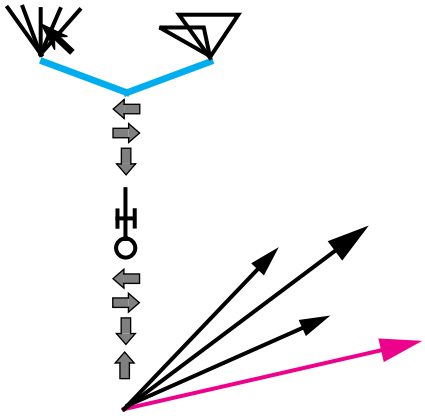


4. Fan Display Chosen.
Trajectories will be displayed as fans. Shown in **5.**

Figure 80: Operation of Ballistics Box



5. Trajectory Fans.
Weapon's possible trajectories are depicted as fans.



6. Line Display Chosen.
Trajectories will be displayed as lines. Shown in **7.**

7. Trajectory Lines.
Weapon's possible trajectories are displayed as lines.

Figure 81: Operation of Ballistics Box(Continued)

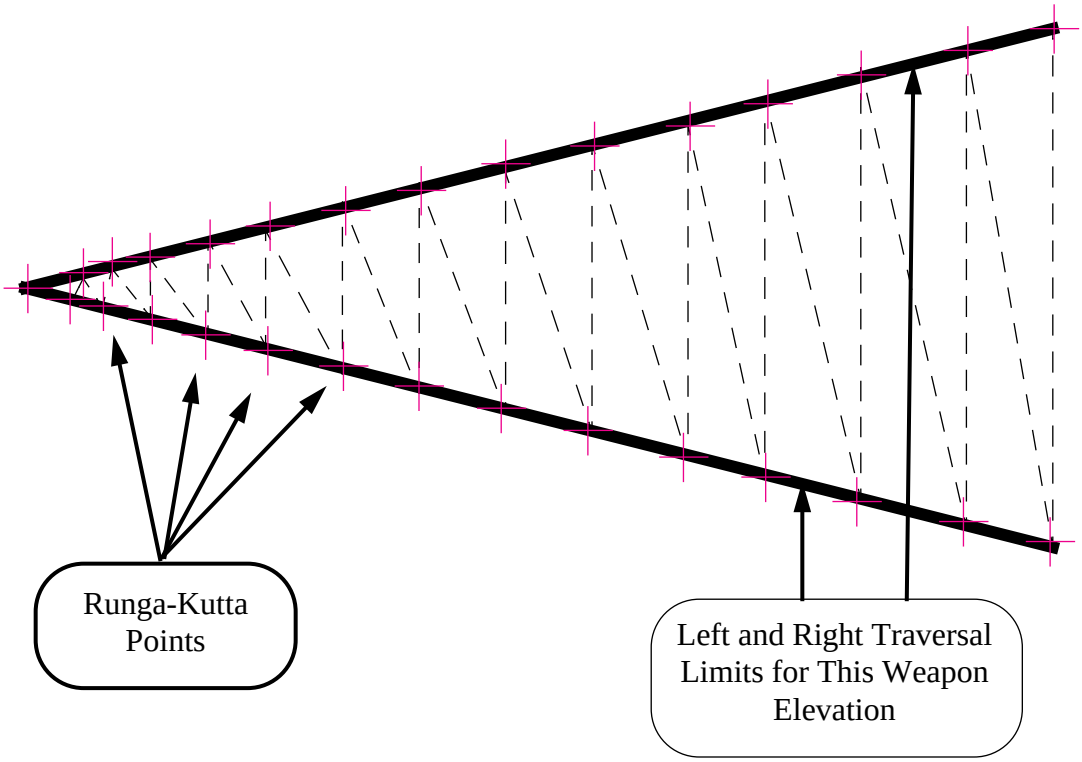


Figure 82: Trajectory Fan Triangle Strip

po_ball.cc. The implementation of the ballistics box will be presented in the same way in which the cone tree menu system was presented in Chapter V, that being C++ code and a dissected diagram of the ballistics box. The C++ code is presented in Figure 83 and the dissected ballistics box is presented in Figure 77.

The structure of the building box actually contains a MENU_LEVEL class object. A different menu is made for each type of weapon selected from the cone tree menu and the type of menu to be used is set in the terminator callback from the cone tree menu. In Figure 77 the top portion of the ballistics box class is shown to actually be a menu. Recalling the structure of MENU_LEVEL (see Figure 70) the base of the menu's Performer structure is

```
class BALL_BOX{
public:
      pfDCS*     root,
                 *lowRightDCS,
                 *upRightDCS,
                 *upLeftDCS,
                 *lowLeftDCS,
                 *checkDCS,
                 *menuDCS,

      float      cumRotLeft,
                 cumRotRight,
                 cumRotUp,
                 cumRotLow,
                 spread;

      pfGroup*   downGroup,
                 *upGroup,
                 *rightGroup,
                 *leftGroup,
                 *HLGroup;

active           current;
ballType         typeBall;
COEF             coef;

BALL_BOX();

};
```

Figure 83: BALL_BOX Class

a pfDCS named `root`. In the case of the ballistics box, the menu's `root` is simply used to place the menu in the correct *relative* position. Also included is the proper icon for the chosen weapon. The menu's `root` is then added as a child to the ballistics box's `root` which as a pfDCS is used to correctly position the ballistics box on the terrain. Again, note the menu's terminators. There is no reason why a terminator has to be a particular icon, rather the actions of selecting an icon are wholly dependant on the associated callback.

`lowRightDCS`, `upRightDCS`, `upLeftDCS` and `lowLeftDCS` each contain a pfGeode which contains the geometry for each of the four "rays" in the ballistics box. Each pfGeode is constructed with the function `makeArrowGeo` which constructs a "ray" using a triangle

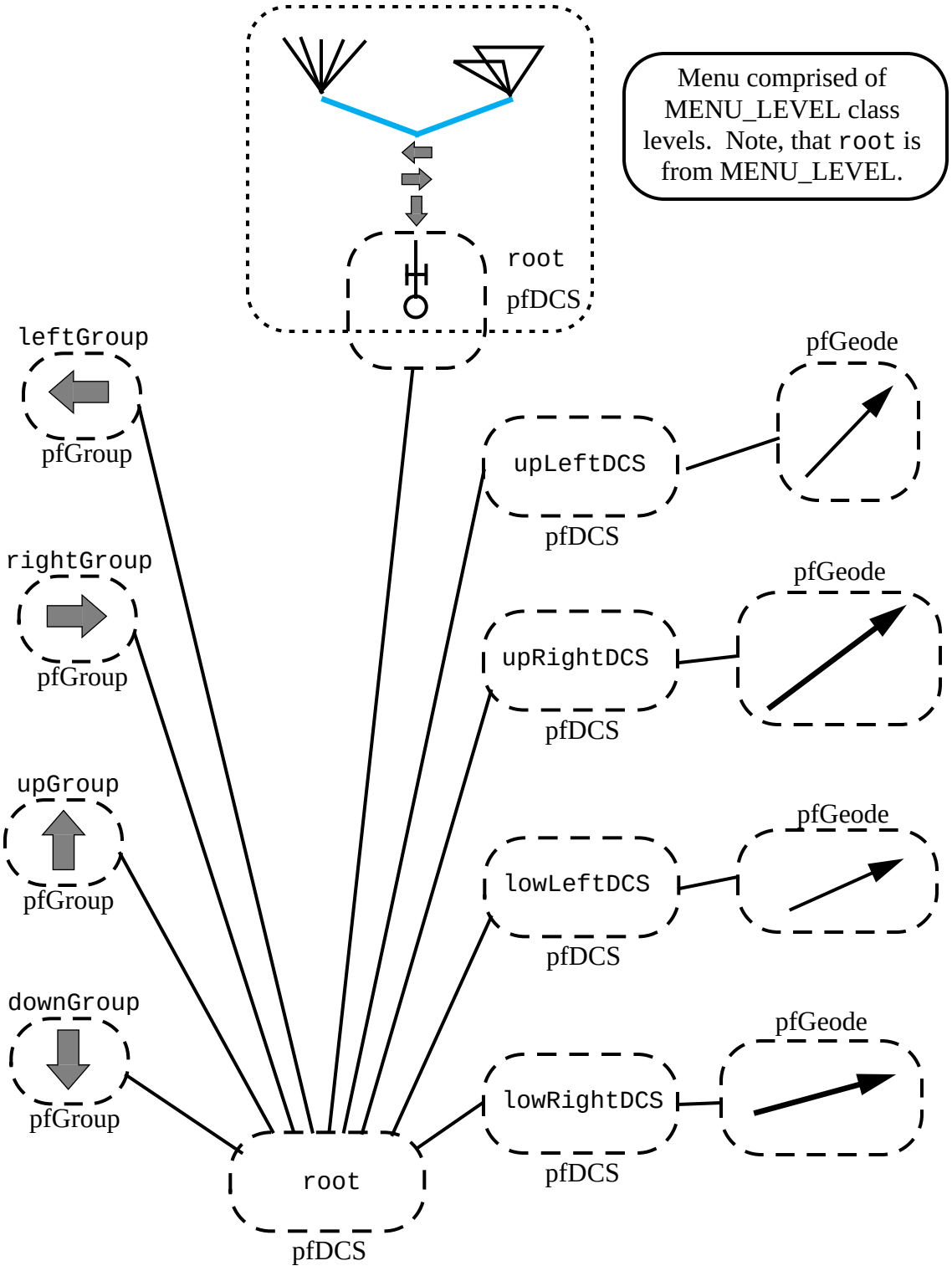


Figure 84: Dissected Ballistics Box

strip. The pfGeodes are actually 3D cylinders and are constructed by reading in vertices in a way similar to `make_meas` (see page 100). However, the redundant Performer structure (see Figure 54) required for a point measure are not required for the “rays”. Rather, only one pfGeode is constructed. Since it is not in a pfBillboard, not below a pfSwitch and not constructed of lines, it can be picked and highlighted using Perfomer functions.

Once the pfGeodes are constructed they are put in the appropriate pfDCSs which enable the rays to be traversed and elevated. Lastly, a callback is assigned to the “rays” in the same manner callbacks were assigned to menu icons (see page 126). The callback highlights the ray (using Performer highlighting) and sets the variable `current`(see Figure 83). `current` contains the ray which is currently being highlighted and manipulated.

downGroup, upGroup, rightGroup and leftGroup are each pfGroups and are constructed and assigned callbacks (see page 126) in the same manner in which the cone tree arrows are constructed. They are assigned callbacks which traverse or elevate the selected ray which is contained in `current`. The callback checks which ray is being manipulated and which action is being done to that ray. Recalling that when rays are moved they move as pairs, the callback finds the appropriate two pfDCSs and rotates them by an amount contained in the members `cumRotLeft`, `cumRotRight`, `cumRotUp` or `cumRotLow`. Each of these members contains the accumulative rotations of the ray pfDCSs. One additional member, `spread`, contains the angular spread between the rays. This is used to ensure that the rays cannot spread greater than the allowed traversal of the weapon and to ensure that “negative traversal” does not occur, for example the lower left ray being rotated further right than the lower right ray. Lastly, each of the above pfGroups and pfDCSs are added as children to `root`.

An overview of the BALL_BOX implementation can now be given. After selecting the trajectory visualizer from the cone tree menu system the BALL_BOX graphical structure of Figure 77 is placed on the terrain by translating `root` to the correct position. The user then selects a ray, which executes a callback which highlights the ray and sets it

as `current`. Next, a BALL_BOX menu arrow is chosen, which by callback rotates the appropriate ray's pfDCS. When the rays have been positioned as desired, the terminator from the menu portion of the BALL_BOX is selected which by callback, collapses the menu, and calculates and displays the trajectories.

C. SUMMARY

This chapter has examined the visualization of weapons trajectories. The motivation for doing the visualization was discussed. It was shown how the trajectory visualizer is selected from the cone tree menu. An operational overview was given for the ballistics box tool. The actual visualization of the trajectories was discussed. Lastly, the implementation of the ballistics box was discussed.

VII. CONCLUSION

A. RESULTS

The Sand Table was constructed and runs on the most current version of NPSNET IV.8. The system is currently functional with the current version of ModSAF which is ModSAF 1.5. To run the system NPSNET is started as normal. The Sand Table system can run using any terrain database which is currently available to ModSAF and NPSNET. Any number of Sand Table or ModSAF stations can be brought up with the PO Protocol "handshaking" being successful. The only user coordination needed is for the users of ModSAF stations to select the "NPSNET" overlay. There was no way to select this overlay remotely from the Sand Table and the most the Sand Table could do was to notify ModSAF of the overlay's existence. After this overlay is selected all measures constructed on either the ModSAF station or the Sand Table will function. Measures constructed on any station will be displayed remotely on any other station. Measures constructed on a Sand Table will be correctly maintained by ModSAF. Measures created on ModSAF can be updated by any Sand Table station.

The cone tree menu system can be positioned on the terrain as shown in Figure 85. From the menu measures can be built and placed on the terrain. In this case a point measure is under construction. Figure 86 depicts a contact point and a coordinating point placed on Fort Benning Terrain. These measures could have been constructed using the cone tree menu system. Alternatively, the measures could have been created at remote stations. They will appear on the local Sand Table and can be manipulated in real time. Figure 87 shows a minefield which was created on a ModSAF station as rendered on the Sand Table. Additionally, there are several point measures and a base level cone tree menu. Also note the tank and helicopter present. This demonstrates that all of the pre-existing functionality of NPSNET is still available. Figure 88 shows the same minefield after having been

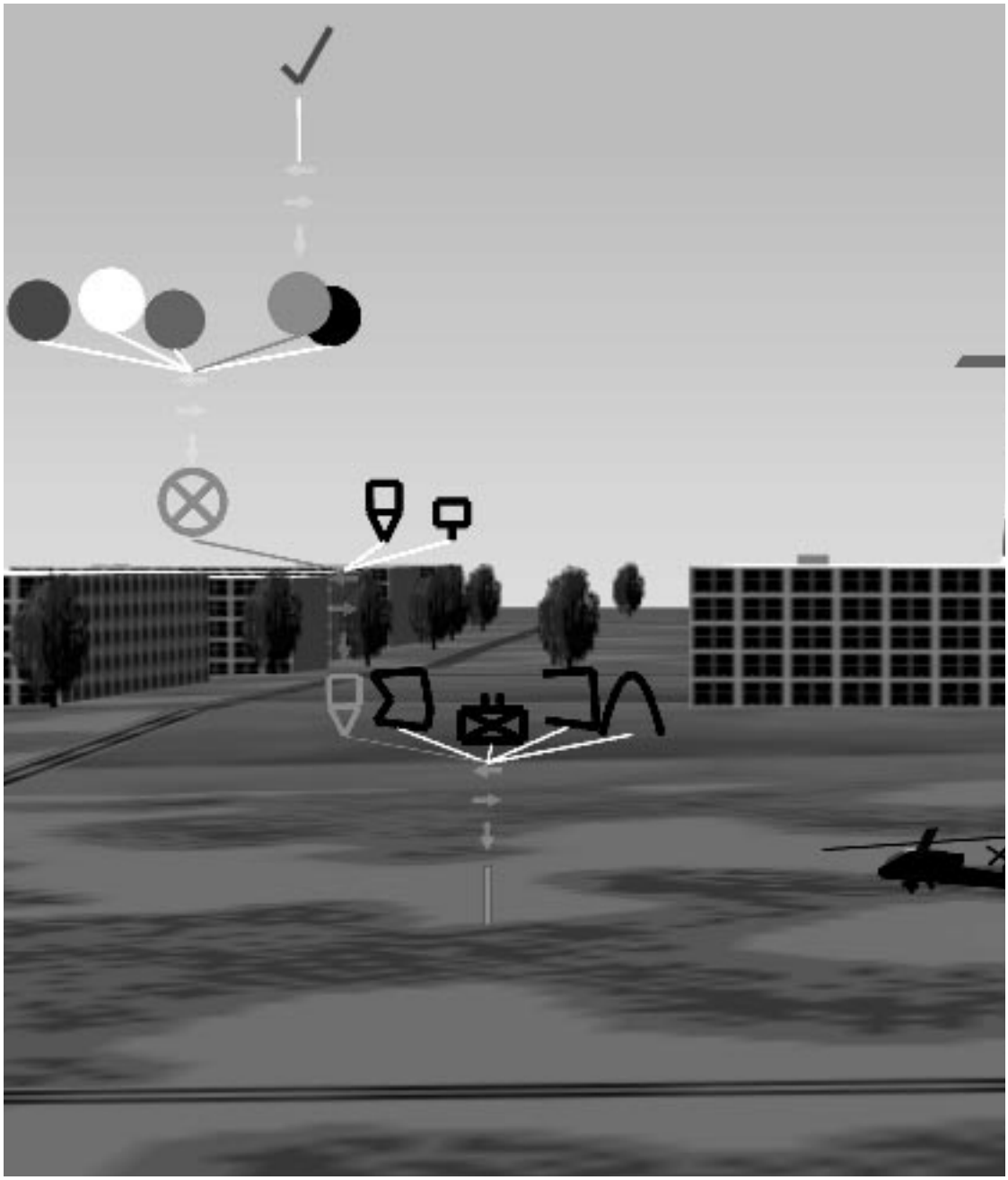


Figure 85: Cone Tree Menu Placed on Fort Benning Terrain

Figure 86: Point Control Measures on Fort Benning Terrain

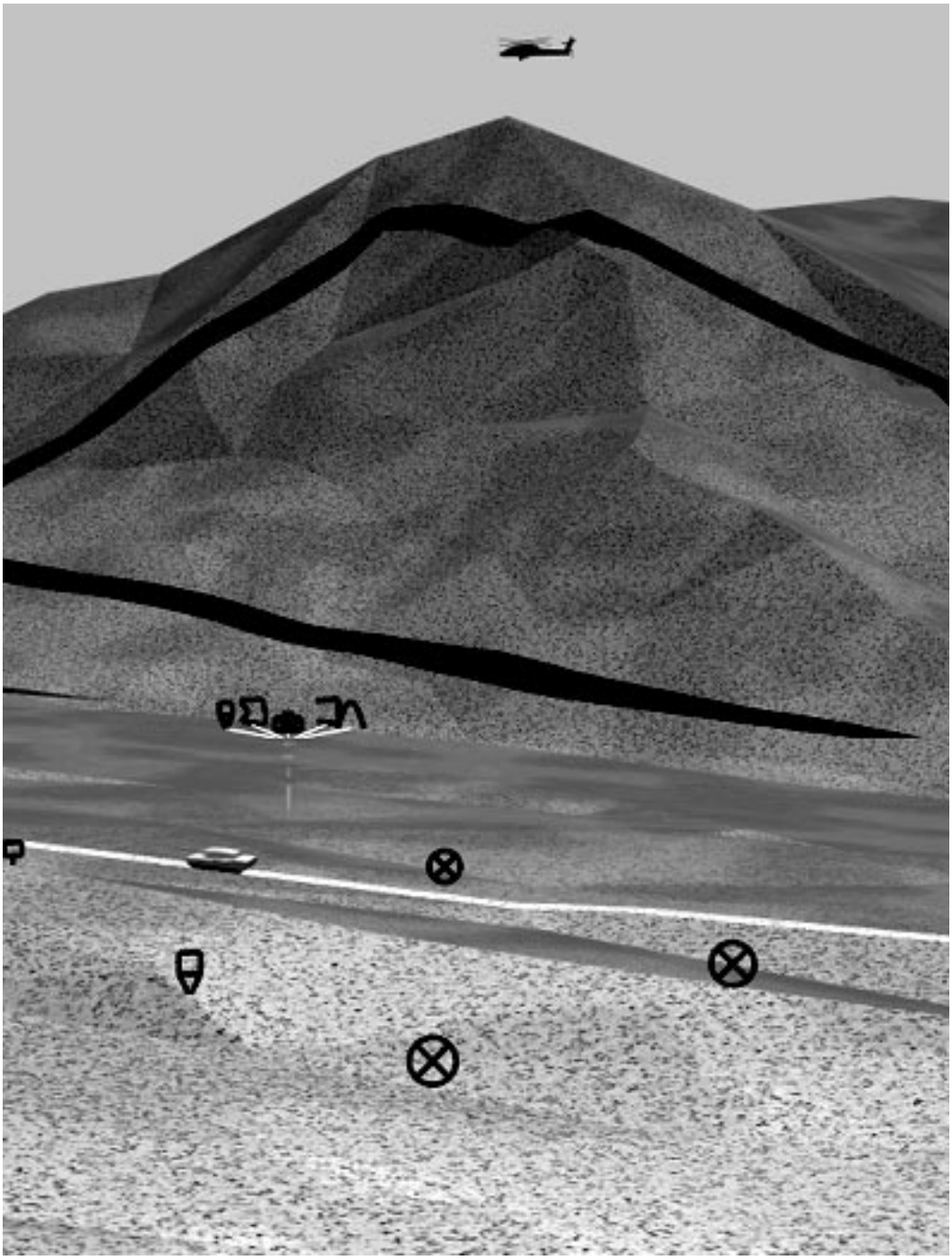


Figure 87: ModSAF Minefield Placed on Range 400 Terrain

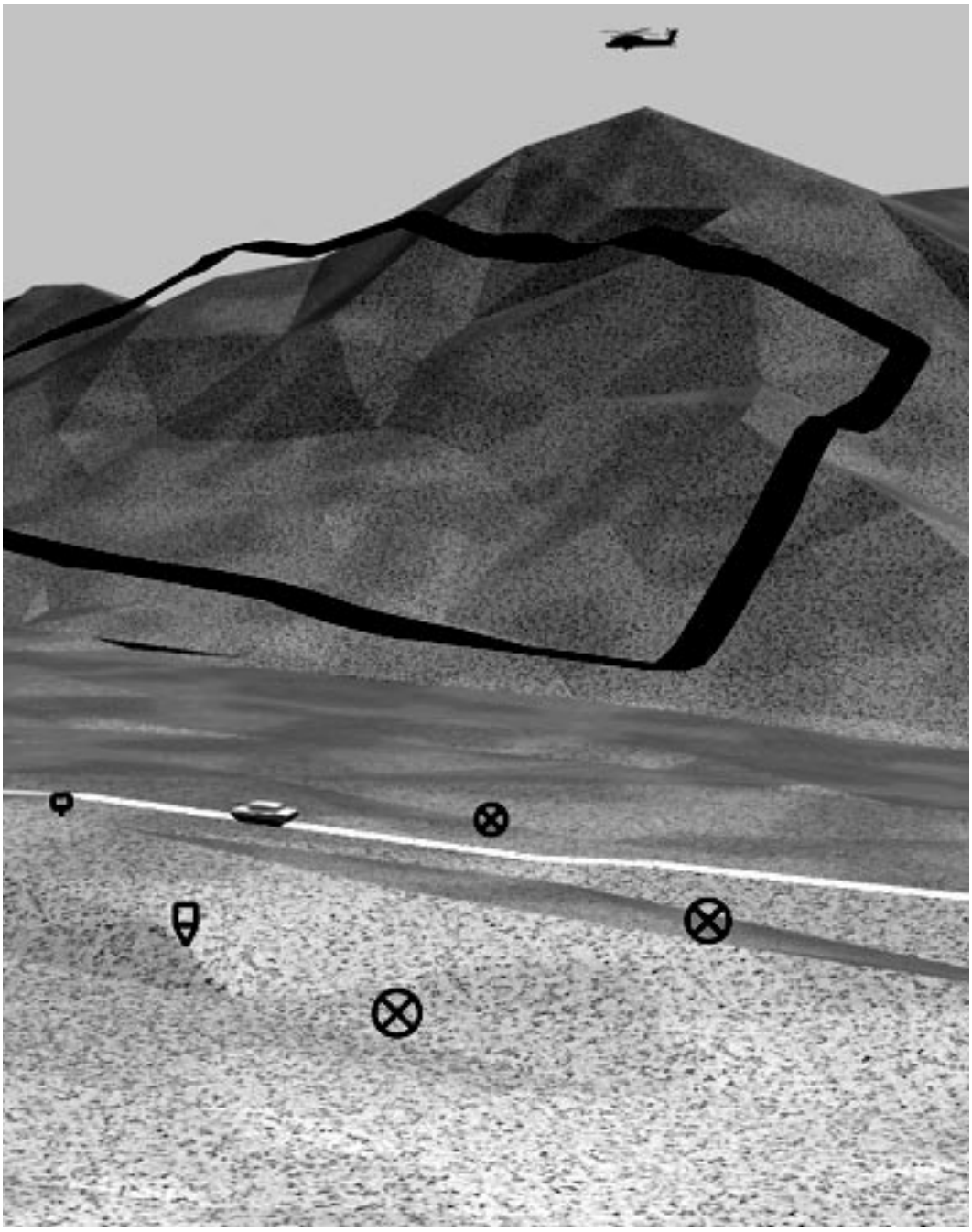


Figure 88: Minefield After Being Dragged and Dropped

dragged and dropped. The changes were then broadcast to the ModSAF station which reflected the change in position. As the minefield was being dragged it was represented with a low resolution model which enables the drag to occur in real time.

Linear control measures can also be built on the Sand Table. Figure 89 shows an open line being built. The cone arrows represent where the line's prospective points are placed. The cone arrows or the entire line can be dragged and dropped as it is being constructed. Figure 90 shows the line after construction is complete.

Similarly, Figure 91 shows a closed line loop being constructed on Range 400. Note, that the lines are terrain following. Figure 92 shows the line loop after construction is complete. Also in Figure 92 is another line created on the Sand Table using a different color.

Lastly, the Sand Table implemented the weapon's trajectory visualizer. Figure 93 shows the weapon's trajectory visualizer tool being selected from the cone tree menu system. In Figure 93 a mortar is being selected from the cone tree menu. Figure 94 shows the rays which represent the desired elevation and traversal of the mortar. Lastly, Figure 95 shows the fans actually representing the projectile flight paths.

B. FUTURE WORK

The Sand Table in the virtual environment succeeded in implementing a system to aid in the visualization of abstract measures during military planning. This was the first effort in implementing a Sand Table and the possibilities for future work are promising. The Sand Table provided an architecture which is very extensible. Listed below are areas in which further work on the Sand Table may prove beneficial.

1. More Robust Measure Representation

The current Sand Table represents three different types of point control measures. This should be increased to depict an inclusive variety of military point symbology. Similarly, linear measures are able to be represented by varying thickness and color of lines; however, only one *style* of line can be represented. The Sand Table should be extended to represent

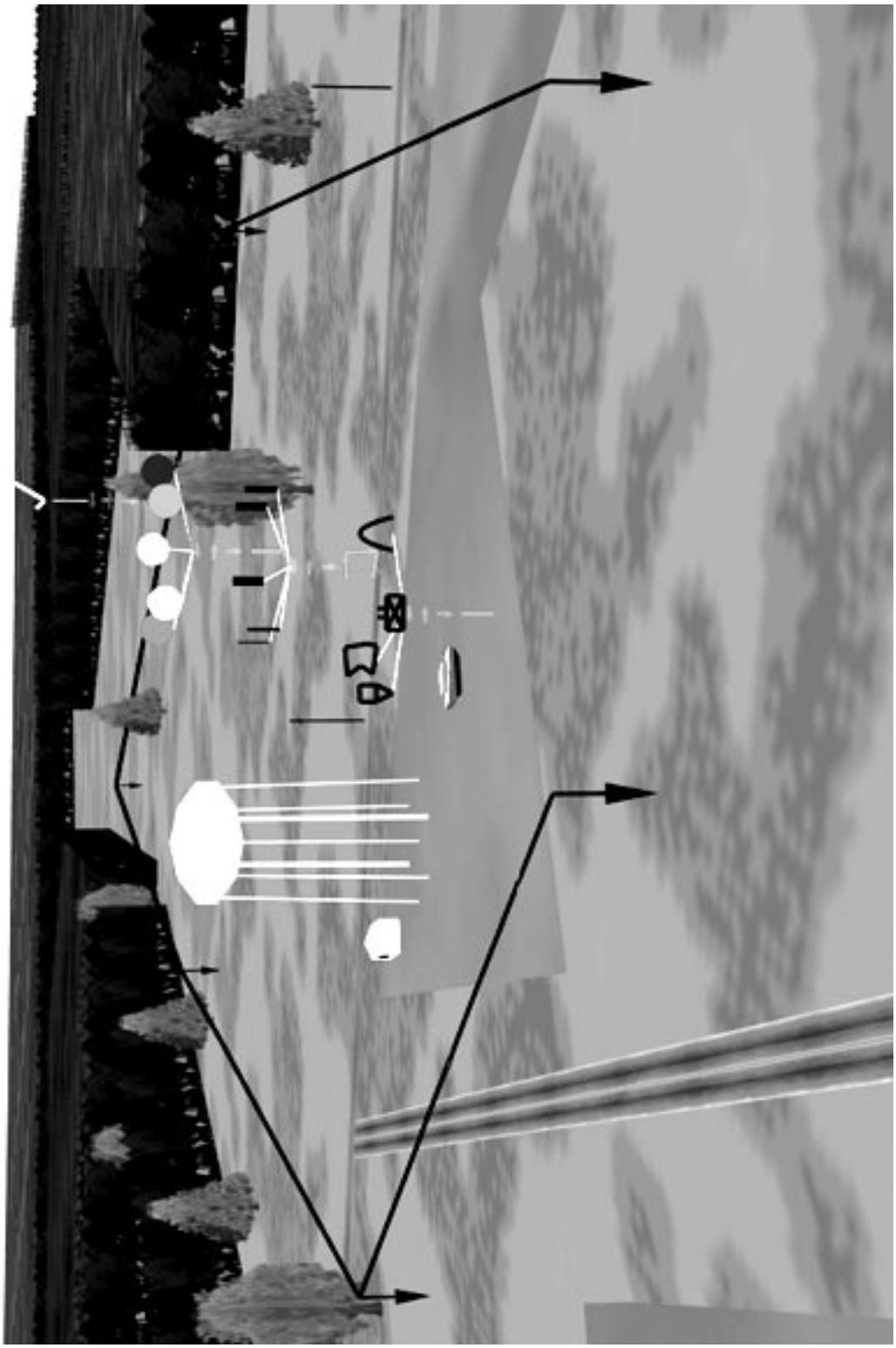


Figure 89: Line Being Built on Fort Benning Terrain

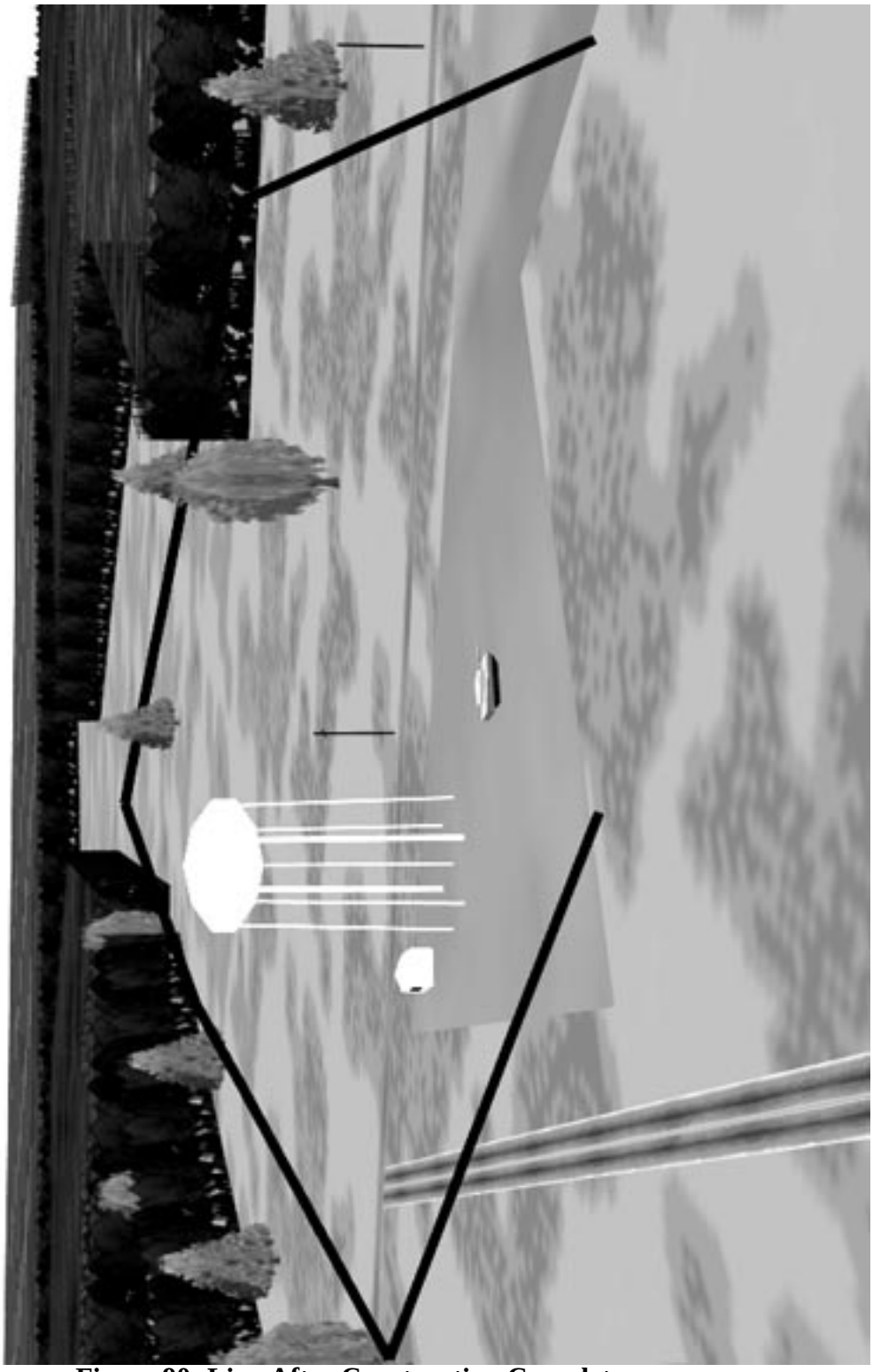


Figure 90: Line After Construction Complete

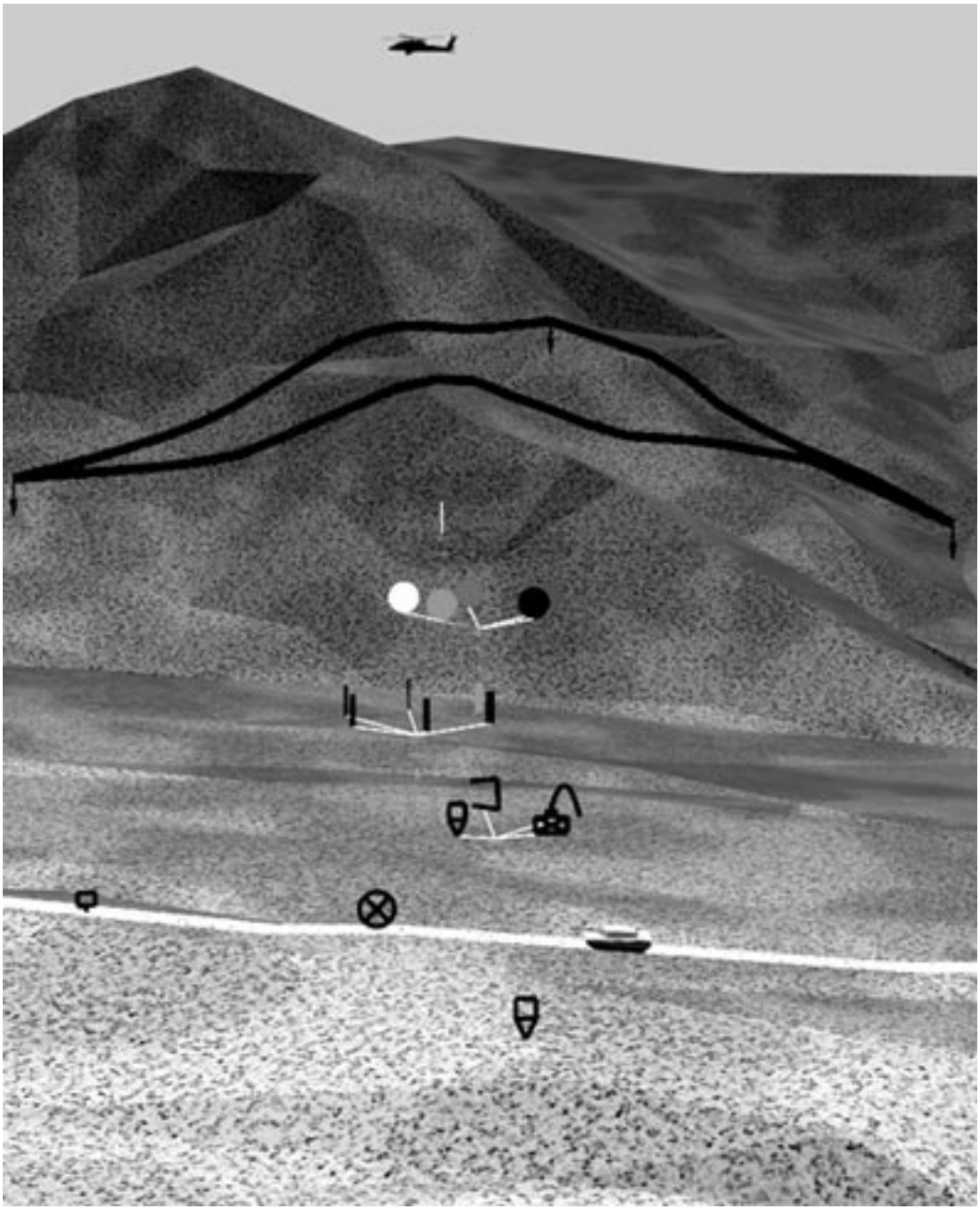


Figure 91: Closed Line Loop Constructed On Range 400

a more inclusive set of military linear measures. Additionally, axes of advance and heliborne axes of advance should be able to be displayed. Lastly, the minefields should have a variety of styles and representations.

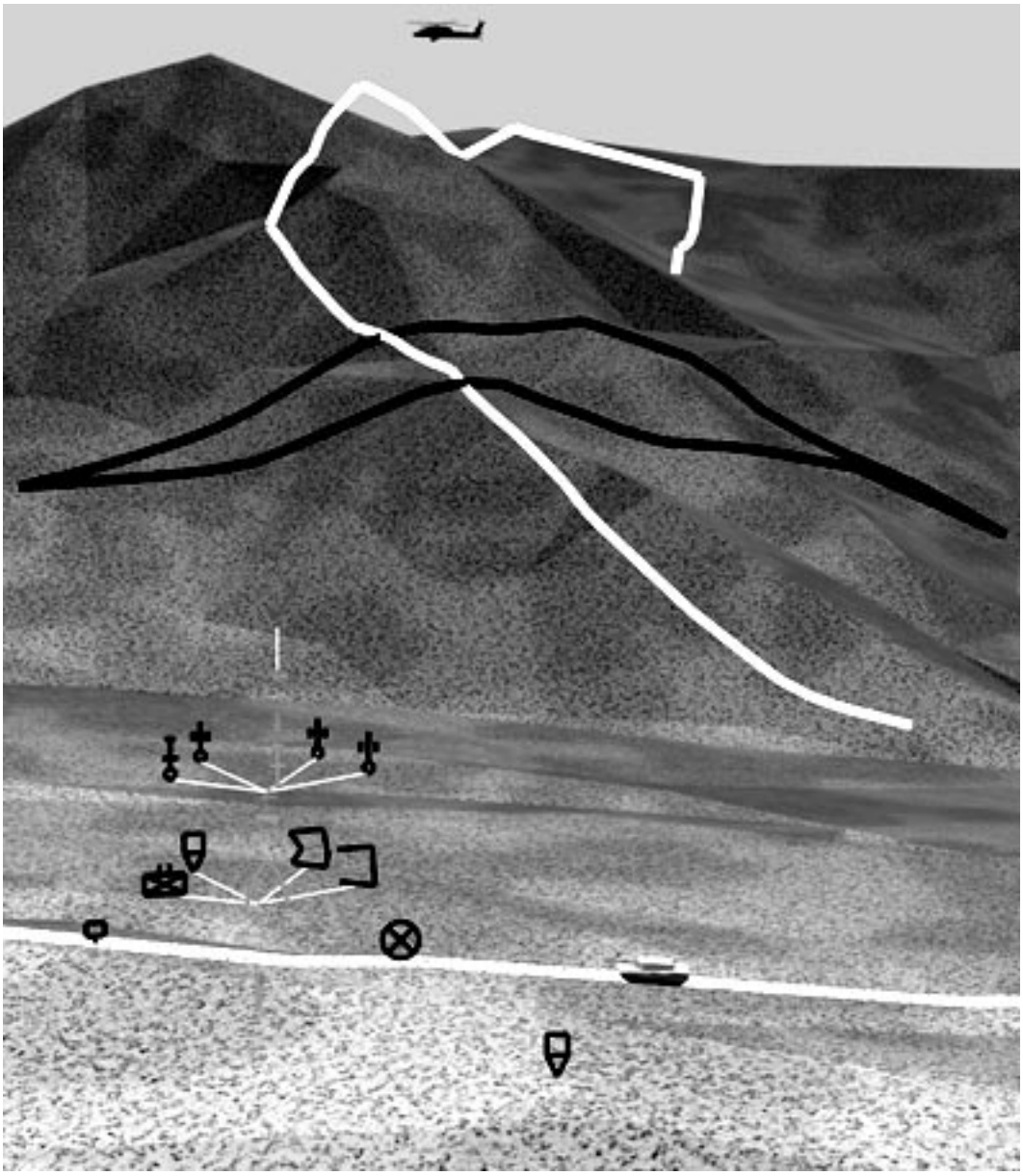


Figure 92: Closed Line Loop After Construction

2. Extended Planning Capabilities

The Sand Table can represent measures on the terrain; yet, it cannot build a complete battle plan and execute the plan. The Sand Table should be extended to utilize more of ModSAF's capabilities concerning the semi-autonomous management of forces. In the

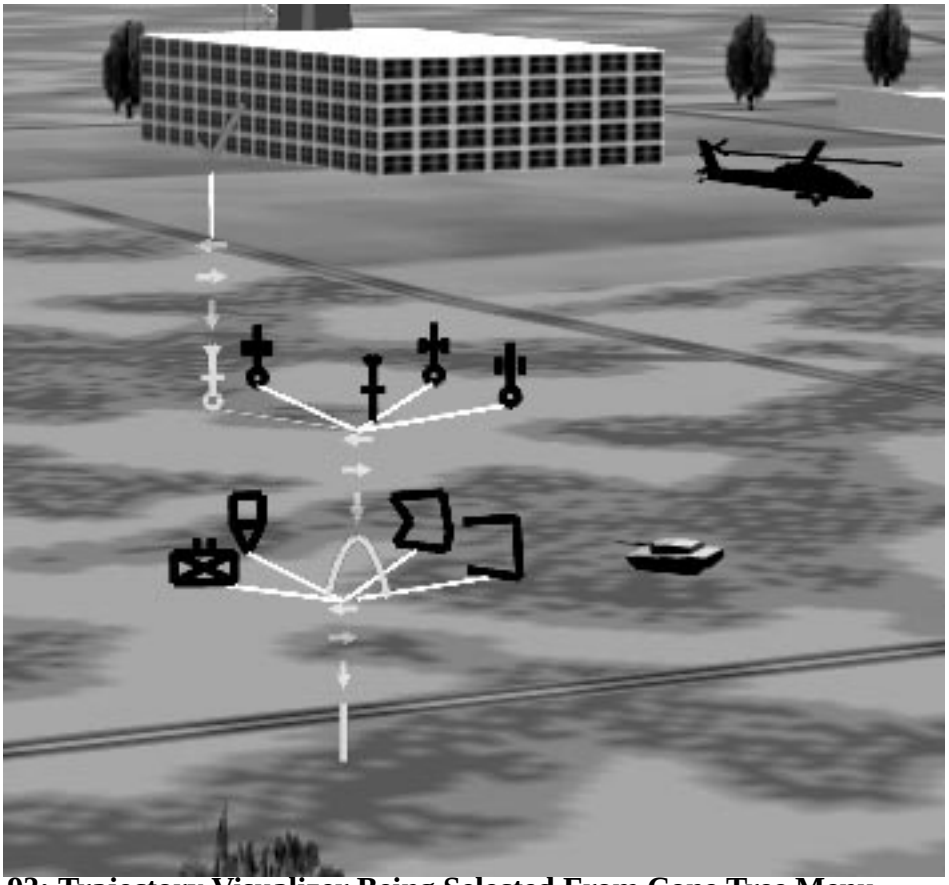


Figure 93: Trajectory Visualizer Being Selected From Cone Tree Menu

future the users should be able to construct a plan and see units moving over terrain with respect to the control measures which were laid down. The current Sand Table can render entities on the battlefield as was seen with the tank and helicopter in the above figures; however, these units are other NPSNET stations simulating vehicles. Their movements are all in real time and they are single vehicles. The planning capability required in the Sand Table dictates a need for a user to be able to plan the movements of entire units. The user should then be able play back plans and make modifications.

With this planning capability, the user should be able to view the units as military symbology *or* as the actual vehicles and equipment which comprise the units. Further, the

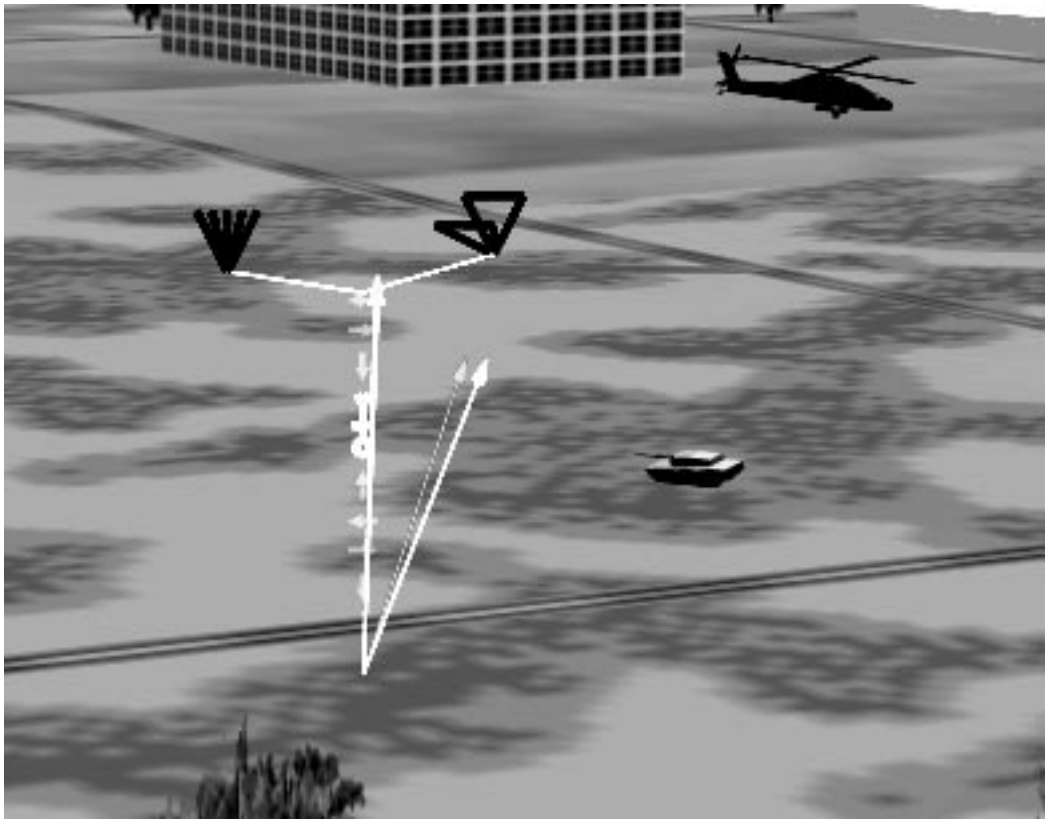


Figure 94: Rays Setting Mortar Firing Area

user should be able to select and change how the units are depicted. For example a tank platoon should be able to be represented as the symbol for a tank platoon or the actual tanks. This planning ability should also include the creation and tasking of units from the cone tree menu system.

3. Additional Object Creation and Manipulation

While the Sand Table can create, drag, and drop control measures, the capability to drag additional objects should be created. For example, the user should be able to create a truck or pile of debris and place it on the terrain. These objects should then be able to be dragged and dropped as control measures are and their movements should be networked as are the control measures.

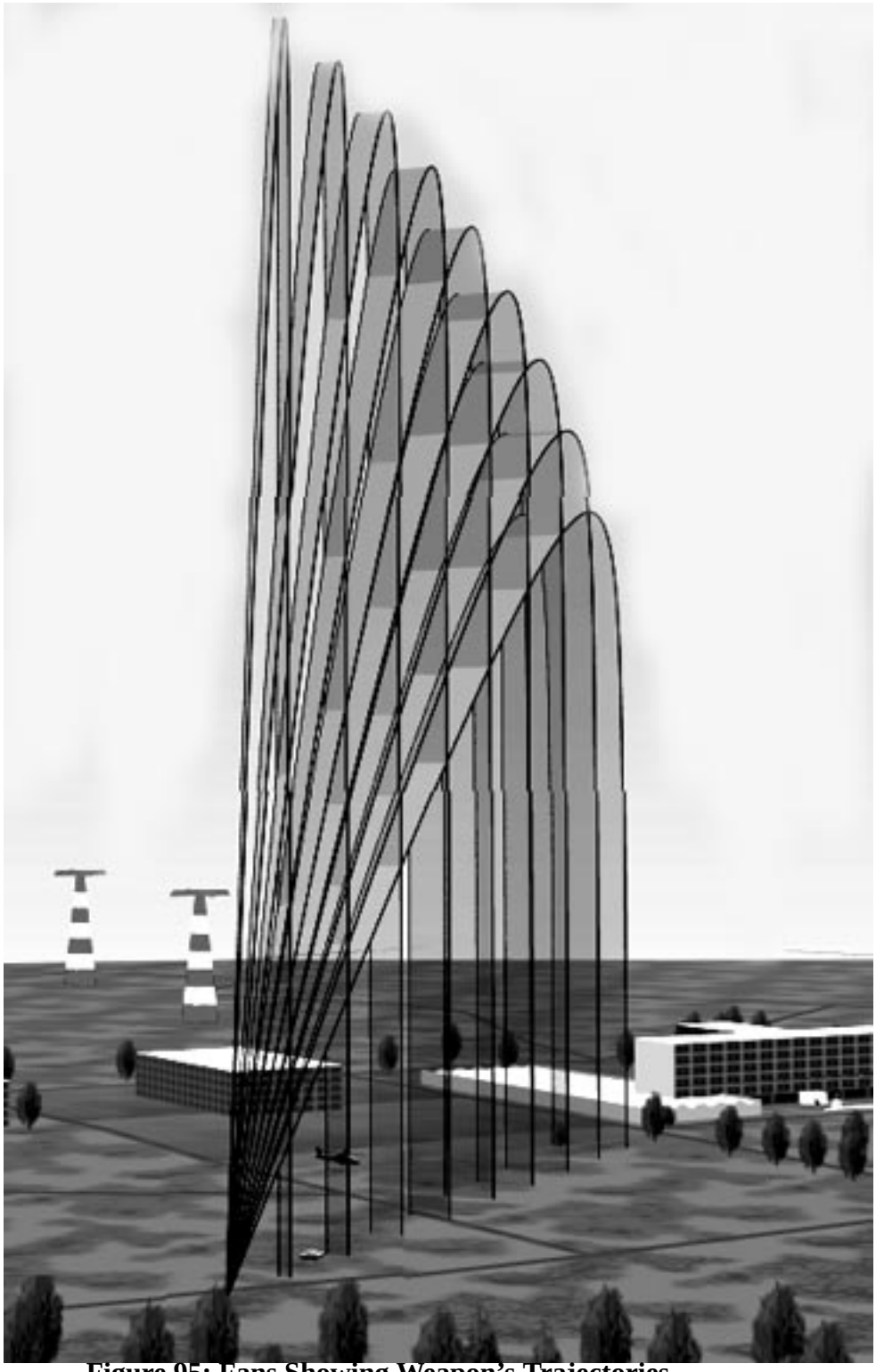


Figure 95: Fans Showing Weapon's Trajectories

Additionally, entities on the battlefield should be able to be dragged and dropped. For example, the tank or helicopter in the above figures should be able to be selected and moved to another location. This has not existed in prior versions of NPSNET because such movement does not make sense in regards to vehicle simulation and game play. It does make sense in rehearsal for example, when a commander may decide in the middle of a battle that a vehicle or unit may be effective in another location. Dragging would provide this capability. Note, that this dragging would entail *much* more effort than the Sand Table, because not only would PO Protocol being used, but the vehicles are actual DIS entities and would have to be moved using the DIS Protocol.

4. Display Research

The Sand Table is currently displayed on normal workstation monitors. Other display technologies should be examined. These include conducting planning with head mounted displays or the "workbench" display used by researchers in Germany. See "Medical Visualization" on page 11. Additionally, stereoscopic applications should be examined with consideration given to different mouse cursor possibilities.

5. Distant Research

While the future work presented above is immediately possible, future work further down the road should also be considered. Such work would include a real time *current* operations version of the Sand Table. The current version of the Sand Table allows planning or training in the virtual world. In the future, the Sand Table could be the display system of what was actually occurring in the real world. For example, during a battle the commander could "view" the battle being fought via a virtual environment. The virtual environment would be provided with data from the real world. The commander would be able to achieve a viewpoint of the battle from a perspective never before seen. A potential danger in this application would be in representing the real world incorrectly in the virtual world. However, the display could provide a means of presenting a large amount of

information which commanders already deal with and are currently organizing and visualizing in their heads.

Lastly, the Sand Table concept could be the basis for an *augmented reality* system for actual units in the field. Graphical displays could be projected on a helmet mounted display indicating control measures. For example a pilot could "see" the route he was assigned to fly superimposed over the physical terrain. Additionally, he could "see" and avoid weapons trajectories which would again be superimposed on physical terrain.

C. SUMMARY

The Sand Table was implemented successfully. It is "up and running" and is under current NPSNET configuration management. It accomplished many of the goals of visualizing abstract military data. It is networked and provides intuitive visualization and manipulation of control measures on virtual terrain. The Sand Table is a good starting point for future development and provides an easily extensible architecture for such efforts.

LIST OF REFERENCES

[BANK95] Banks, Davis C. and Kelley, Michael, "Tracking a Turbulent Spot in an Immersive Environment," *Proceedings 1995 Symposium on Interactive 3D Graphics*, pp. 171-172, ACM Press, 1995.

[BARH94] Barham, Paul, T., Pratt, David, R., Zyda, Michael, J., Locke, John and Falby, John, "NPSNET-IV: A DIS Compatible, Object-Oriented, Multiprocessed Software Architecture For Virtual Environments," Naval Postgraduate School, Monterey, CA, 1994.

[BRYS91] Bryson, Steve and Levit, Creon, "The Virtual Windtunnel: An Experiment for the Exploration of Three-Dimensional Unsteady Flows," RNR Technical Report RNR-92-013, Moffet Field, CA, 1991.

[DRIS95] Driskill, Elena and Cohen, Elaine, "Interactive Design, Analysis, and Illustration of Assemblies," *Proceedings 1995 Symposium on Interactive 3D Graphics*, pp. 27-33, ACM Press, 1995.

[KALA93] Kalawsky, Roy, S., *The Science of Virtual Reality and Virtual Environments*, pp. 313-318, Addison-Wesley, 1993.

[KIJI94] Kijima, Ryugo, Shirakawa, Kimiko, Hirose, Michitaka and Nihei, Kenji, "Virtual Sandbox: Development of an Application of Virtual Environments for Clinical Medicine," *Presence Teleoperators and Virtual Environments*, vol. 3, no. 1, pp. 45-59, MIT Press, 1994.

[KRUG94] Kruger, Wolfgang, Bohn, Christian, A., Frohlich, Bernd, Schuth, Heinrich, Strauss, Wolfgang and Wesche, Gerold, "The Responsive Workbench A Virtual Environment for Scientists, Engineers, Physicians and Architects," Sankt Augustin, Germany, 1994.

[MCCO87] McCormick, B., T.A. DeFanti, and M.D. Brown, eds, "Visualization in scientific computing," *Computer Graphics*, vol 21, no. 6, 1987.

[MOHN94] Mohn, Howard, Lee, "Implementation of a Tactical Mission Planner for Command and Control of Computer Generated Forces in ModSAF" (Master's Thesis, Naval Postgraduate School, Monterey, California, 1994).

[NRC95] National Research Council, *Virtual Reality Scientific and Technological Challenges*, pp. 433-437, National Academy Press, 1995.

[ROBE91] Robertson, George G., Mackinlay, Jock, D., and Card, Stuart, K., "Cone Trees: Animated 3D Visualizations of Hierarchial Information," Xerox PARC, Palo Alto, CA, 1991.

[ROHR94] Rohrer, Jim, J., Design and Implementation of Tools to Increase User Control and Knowledge Elicitation in a Virtual Battlespace," Master of Science Thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, 1994.

[SAFF93] Saffi, Maureen, R., The ModSAF Interface, Version B, User Guide, Loral Advanced Distributed Simulation, Inc., 1993.

[SMIT93] Smith, Joshua E., Courtemanche, Anthony, J., "LibPO, Persistent Object Library," ModSAF B Software Documentation, Loral Advanced Distributed Simulation, Inc., 1993.

[STAS92] Stasko, John, T., "Three-Dimensional Computation Visualization," Technical Report GIT-GVU-92-20, Georgia Institute of Technology, Atlanta, GA, 1992.

[STYT95] Stytz, Martin, R., Hobbs, Bruce, Kunz, Andrea, Solz, Brian and Wilson, Kirk, "Portraying and Understanding Large-Scale Distributed Virtual Environments: Experience and Tentative Conclusions," *Presence*, vol. 4, no. 2, pp. 146-168, MIT Press, 1995.

[ZYDA93] Zyda, Michael J., Pratt, David R., Falby, John S., Barham, Paul T. and Kelleher, Kristen M., "NPSNET and the Naval Postgraduate School Graphics and Video Laboratory," Presence, Vol. 2, No. 3.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 013
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr David R. Pratt, Code CSPr 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Mr. John Falby, Code CSFa 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Mr. Paul Barham, Code CS/Barham 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

7. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

8. Commanding Officer 3
   NCCOSC RDTE Division
   San Diego, CA 92152-5000
   Attention: Jeff Clarkson, Code 44206

9. Captain Samuel A. Kirby.....2
3424 Chambersburg Avenue
Duluth, MN 55811